University of Alberta

Library Release Form

Name of Author: Maria Cutumisu

Title of Thesis: Multiple Code Inheritance in Java

Degree: Master of Science

Year this Degree Granted: 2003

University of Alberta

Multiple Code Inheritance in Java

by

Maria Cutumisu ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2003

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Multiple Code Inheritance in Java** submitted by Maria Cutumisu in partial fulfillment of the requirements for the degree of **Master of Science**.

To my family, always supportive.

Abstract

Java has multiple inheritance of interfaces, but only single inheritance of code. This situation leads to code being duplicated in Java library classes and applications. We describe a generalization of a *Java Virtual Machine* (JVM) to support multiple inheritance of code.

Our approach places code in interfaces, without requiring language syntax changes or compiler modifications. In our extended JVM, we use interfaces to represent either new types of interfaces with code or traditional interfaces in Java. We define and implement a super call mechanism resembling the one in C++, in which the programmer can specify an inheritance path to the desired superinterface implementation. We introduce a simple notation for super calls to interfaces. Furthermore, we develop scripts that allow a programmer to use multiple code inheritance with existing Java compilers.

We have modified a JVM to support multiple code inheritance. Our implementation does not affect the running time or the semantics of standard single inheritance Java programs and executes correctly programs that use multiple inheritance.

Acknowledgements

During my studies at the University of Alberta, I benefited from the help of my supervisors, colleagues, and friends.

I would like to express my gratitude to my supervisors, Duane Szafron and Paul Lu, for the privilege of working with them and for the effort they invested in this research. Their support, enthusiasm, and constant encouragement resulted in an excellent environment for my research, motivating me to strive towards meeting their standards and expectations.

I would like to thank Nelson Amaral and Dennis Ward, also members of my examining committee, for their valuable suggestions that improved my dissertation.

I am grateful for having the opportunity to study at the University of Alberta, which enabled me to meet my first friends in Canada in the Computing Science department.

I would especially like to thank Steve MacDonald, Matt McNaughton, and Chris Dutchyn for being very helpful every time I needed advice during my research.

I would like to acknowledge Anita Petrinjak, Adi Botea, and Akihiro Kishimoto (Kishi), my friends from the Computing Science department, always with a great sense of humor and with whom I shared the experience of studying in a completely different environment than in our native countries.

In the same department, I had the chance to meet Luiza Antonie and Alex Coman, my best friends, always helpful, cheerful, and optimistic. I would like to thank them for their care, understanding, and patience, especially during my thesis writing.

I would also like to thank Leo Mocofan, Ernie Novillo, Adrian Driga, John Anvik, Cam Macdonell, Kai Tan, Peng Zhao, James Redford, Calvin Chan, and Dominique Parker for creating a pleasant environment in the software systems laboratory. There are many other people who had a positive influence on this dissertation. I did not forget them and I would like to thank them all as well.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Object-oriented programming languages are tools intended to clearly express the powerful features that define the object-oriented programming paradigm, in an attempt to better model real-world phenomena. Among features such as encapsulation, polymorphism, and inheritance, the latter distinguishes itself as one of the most important mechanisms for organizing, building and reusing *types* in a programming environment. In the absence of inheritance, *types* are independent and they are constructed without taking advantage of possible commonalities; the programmer has to explicitly ensure eventual consistency among similar *types*. Before introducing the concept of multiple code inheritance, we explain the notions of *type* and *inheritance*.

In general, the term **type** is used to describe a set of possible values that obey certain imposed rules (i.e., contain common features, such as a set of operations). Therefore, a type consists of two notions: *value* (or state) that varies across instances of the type and a set of common *operations* for the type. When a variable of a given type is declared, the variable is expected to behave in a certain manner according to the type it belongs to. For example, the mathematical notion of *integer* assumes a specific set of operations for all its values. The binary operation + sums two integers and returns an integer as the result of the computation. The notion of *string* (series of characters) differs from integer in both its set of operations and the nature of its stored data. The binary operation + has the same syntax as the addition operation for integers, but it has a different semantics (i.e., concatenation). Moreover, there are op-

erations for one type which do not make sense for another type: the operation which returns a character at a given position in a string, `charAt(index)`, has no analog in the integer type.

In order to understand *inheritance* [2], some of the underlying concepts of object-oriented languages have to be defined: *objects, classes, interfaces,* and *messages.* Their interactions help programmers to model various real-world situations in software applications.

An *object* is a conglomerate of *behavior* (set of operations) and *data* (state). The data of an object, represented by variables, can be modified through behavior, represented by methods (each method can be further split into a method signature and a method body). Objects interact with each other by passing *messages* which, depending on the type of their receiver object (and possibly on the type of arguments), can trigger specific method executions. A *class* can be seen as a prototype of all objects with the same type of variables and behavior. An *interface* is a contract containing only method signatures and constant declarations. Each class implementing an interface has to meet the requirements of the contract: it has to eventually implement (i.e., provide method bodies for) all the methods declared in the interface it implements. Unlike classes and interfaces, primitive types are non-object types. Since we are interested in the mechanism of object inheritance, we will ignore the primitive types in our discussion below.

We use the term **property** (or **feature**) of an object to refer to any combination of *method signatures*, *method bodies*, or *data* for that object. *Method signatures* (operations or prototypes) constitute all the messages that can be sent to an object of a type, *method bodies* (code) are the methods that provide an implementation for the signatures defined in that type, and *data* (instance variables, state, or attributes) represent the information stored, not computed, in the object. Java uses the interface and class language constructs to group objects with this variety of properties. Interfaces are groups of method signatures. Classes consist of method bodies and data.

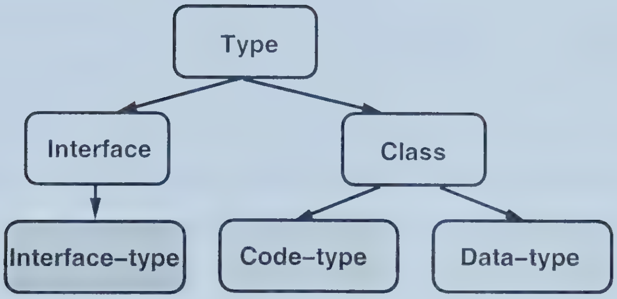


Figure 1.1: Type distinction in Java.

1.0.1 A View of Object Types

In order to better individualize, separate, and exploit these three kinds of properties, we introduce three new language constructs: **interface-type** defines the *operations* for a group of objects, **code-type** associates *code* with each operation of the interface-type that it implements, and **data-type** describes the *data* representation (the data layout of objects that implement code-types for an interface-type) and supports object creation.

It is desirable to design and implement software that explicitly differentiates among these concepts. The motivation for and advantages of separate language mechanisms for these concepts are described by Leontiev, Ozsu, and Szafron [19] [20]. Unfortunately, most popular object-oriented programming languages do not entirely separate these three concepts. For example, Java has two language constructs, *interface* and *class*, that partially separate these three concepts (as shown in Figure 1.1), whereas Smalltalk and C++ combine all three concepts into a single *class* construct. Brad Cox deliberately did not provide the Objective-C programming language with multiple inheritance because he believed that, while inheritance was an implementation tool, it alone was of little help in specifying classes, both statically (how they fit into their environment) and dynamically (what tasks they can actually perform). As it was defined, inheritance did not alleviate "the lack of robust specification tools for software" [6].

Our long-term goal is to provide separate language mechanisms for each of

these concepts (Figure 1.1). Our short-term strategy is to explicitly model the three separate concepts in existing popular programming languages to evaluate the utility of concept separation and to increase the demand for separation in future languages. This dissertation describes a successful attempt to explicitly model these three concepts as separate language constructs in Java, using existing language constructs.

1.0.2 A View of Inheritance

The term **sub-type** describes any specialization of a type and is represented by an arrow in a class, interface, or primitive type (parent type or super-type) diagram. Sub-types can modify properties of super-types and can also add new ones. However, a sub-type cannot remove a property from a super-type.

When objects of different types (interface-type, code-type, or data-type) have common features, *inheritance* [33] provides a mechanism to reuse some features from a type in another type. It also organizes and builds new types based on existing ones, reducing the number of declarations and the amount of executable code that must be written.

With respect to the number of possible direct super-types of a type in an inheritance diagram, two kinds of inheritance are distinguished: *single inheritance* and *multiple inheritance*. *Single inheritance* allows a type to have at most one direct super-type. *Multiple inheritance* allows a type to have more than one direct super-type, so that the child type represents a combination of features from two or more parent types. In C++, "the original and fundamental reason for considering multiple inheritance was simply to allow two classes to be combined into one in such a way that objects of the resulting class would behave as objects of either base class"[31]. A classic example of multiple inheritance is illustrated in Figure 1.2 and can be found in the standard `iostream` library in C++. An object of class `iostream` is both an `istream` and an `ostream`, because it provides functionality to perform `input` and `output` operations with a stream. Moreover, except for the constructor and destructor, `iostream` inherits all its operations from its parent classes `istream` and `ostream`.

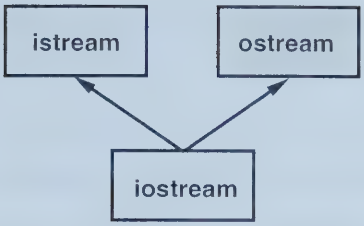


Figure 1.2: Example of multiple inheritance in C++.

There are three distinct useful ways to perceive inheritance in object-oriented programs: **interface inheritance, code inheritance,** and **data inheritance**.

First, we use the term **interface inheritance** to denote the situation when a sub-type inherits the *operations* of its super-types. The principle of *substitutability* states that if a language expression contains a reference to an object whose static type is A, then an object whose type is A or any sub-type can be used instead. *Interface inheritance* relies only on substitutability and does not imply that code or data are inherited. Java uses an interface (Figure 1.1) to implement the concept we have called an *interface-type.* With this terminology, Java currently supports *multiple interface-type inheritance* or *multiple interface inheritance.*

Second, we use the term **code inheritance** when a *code-type* reuses the binding between an operation and the associated code in its parent's *code-type.* *Code inheritance* can be used independently of data representation since there are many operations that can be implemented by simply calling more basic operations. Each object-oriented language implements *code-types* in its own way. In Java, C++ and Smalltalk, a *class* is used as a *code-type.* However, in all three languages, classes have two other responsibilities, namely data representation and object creation. In C++ and Smalltalk, the class also has the interface-type responsibilities that are done in Java interfaces. Java and Smalltalk have only single *code inheritance,* but C++ has multiple *code inheritance* through classes. In this dissertation, we show a novel way to implement *multiple code inheritance* in Java. This is the essential step to meet

our goal of modeling each of these three concepts separately in Java using existing language constructs.

Third, we use the term **data inheritance** when a sub-type reuses *data* (not code) from the super-type. *Data inheritance* allows a *data-type* to reuse the object layout of a parent *data-type.* Of course, classes in Java, C++ and Smalltalk have both the data layout and object creation responsibilities. Unfortunately, they also have other responsibilities that are better suited to the other two language mechanisms that we have called interface-types and code-types. Neither Java nor Smalltalk supports multiple data inheritance, but C++ does.

Since popular programming languages combine *code and data*, they either support both multiple code inheritance and multiple data inheritance (C++), or single code inheritance and single data inheritance (Java and Smalltalk). We use the term **implementation inheritance** to refer to combined *code and data inheritance.* The term **implementation-type** is used for a construct that combines a code-type and a data-type.

1.0.3 Types in Practice

In Chapter 2, we describe how existing programming languages without multiple code inheritance use different alternatives to share code from several types. They all suffer from one or more of these problems: repeated code that bloats the code-base, mistakes when copying similar code, an increased delegation overhead by sending too many messages, and a requirement that all source code must be available. The separation of inheritance concepts is also compromised. For example, in certain situations both interface-type and code-type access is necessary for the programmer to modify the code. Multiple code inheritance, on the other hand, simplifies the work of the programmer, supporting simple definitions of complicated models. Languages such as C++, Clos, Cecil, and Dylan benefit from using this concept.

In the process of analyzing the separation of inheritance concepts as applied to Java, we explored several possibilities in order to achieve multiple code inheritance. One option is to represent code-types by *abstract classes.*

However, the example from Figure 1.2 illustrates that it is often necessary to inherit code from multiple code-types. If code-types were represented as *abstract classes*, we would need to modify Java so that an abstract class can inherit from multiple superclasses.

On the other hand, if we use interfaces to represent code-types, we can take advantage of Java's current multiple-inheritance of interfaces. The problem is simplified to modifying Java to support code in interfaces. We solved this problem by making straightforward and localized changes to the Java Virtual Machine (JVM).

Our approach accesses code in superinterfaces and superclasses using the same inheritance mechanism. We do not support multiple data inheritance, since data cannot be declared in interfaces. However, as will be shown in the next Chapter, multiple data inheritance is the cause of many complications in the implementation of multiple-inheritance in C++. At first glance, it may appear that the opportunities for multiple code inheritance without multiple data inheritance are few. However, as the examples throughout this dissertation show, that is not a concern: all references to data are replaced by abstract accessor method invocations, that are implemented down the hierarchy in data-types (concrete classes).

Our implementation has several advantages: it facilitates code re-use, it supports separation of inheritance concepts, and it improves expressiveness and clarity of implementation.

1.1 Research Contributions

The research contributions of this dissertation include:

1. The first implementation of multiple code inheritance in Java is provided. It is based on the novel concept of adding code to a new type of interface called a code-type. Only straightforward and localized modifications are made to the JVM to support code within the interfaces. All existing programs continue to work as before and suffer no performance penalties. No changes need to be made to the syntax of Java to use multiple

code inheritance, so no compiler changes are necessary. However, syntax changes that would simplify coding are proposed for the future.

2. We show how multiple code inheritance reduces the amount of identical and similar code (such as in the standard libraries) to simplify program construction and maintenance.

3. We have also defined and implemented a super call mechanism that resembles the one in C++, in which programmers can specify an inheritance path to the desired super implementation. We have introduced a simple notation for these super calls that does not require compiler support and proposed a simple syntax for future compiler support.

1.2 Dissertation Organization

In Chapter 2, we review the current state of multiple inheritance. In Chapter 3, we describe the current implementation of those parts of the JVM that are involved in method dispatch. In Chapter 4, we describe how we modified the JVM to support code in interfaces and how this code is dispatched. This idea is the key to our implementation of multiple code inheritance. In Chapter 5, we describe the changes necessary to support a generalization of the super operation for multiple inheritance. In Chapter 6, we describe the experiments we conducted to validate our approach. In Chapter 7, we discuss the mechanism that the programmer uses to apply multiple code inheritance and propose future syntax changes to simplify this mechanism. Finally, in Chapter 8 we present future work and provide a summary.

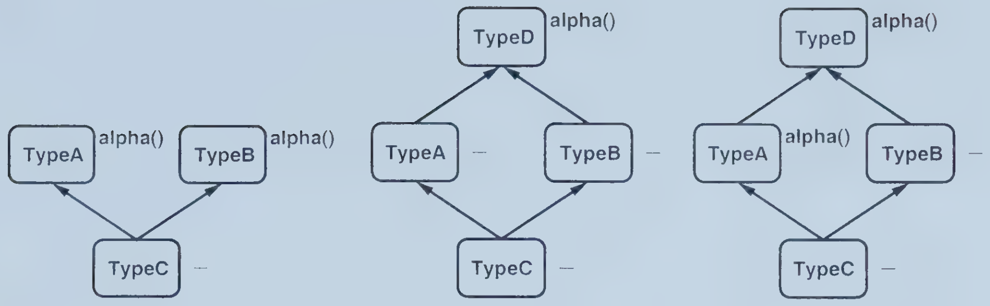
Chapter 2

The State of Multiple Inheritance

This Chapter describes the state of multiple inheritance from four different perspectives. First, it presents some of the problems associated with multiple implementation inheritance which have resulted in its absence from many programming languages. Second, two modalities are described as substitutes for multiple inheritance. Third, the advantages of using multiple inheritance are listed. Finally, a summary of how various programming languages that provide multiple inheritance cope with the issues introduced by multiple inheritance is presented at the end of this Chapter.

2.1 Problems with Multiple Implementation Inheritance

When migrating from single to multiple implementation inheritance, new issues arise due to the existence of several potentially unrelated parents (super-types) from which a child (sub-type) inherits. It may be difficult to determine which particular version of an intended common feature will be propagated from a super-type to a sub-type, if the sub-type does not provide a corresponding feature of its own. Access to each feature from super-types is checked for ambiguity – a situation in which an expression used to access a property from the super-type may not properly differentiate the contributing parent. Five major problems in ambiguity due to multiple implementation (code and data)



(a) Simple ambiguity.

(b) Diamond ambiguity.

(c) Special case ambiguity.

Figure 2.1: Operation ambiguities.

inheritance are analyzed separately, followed by our solution to each of them. The problems are illustrated using examples from C++ [12], a language that supports multiple implementation inheritance.

2.1.1 Problem 1: Operation Code Ambiguity

Figure 2.1(a) illustrates the case in which different code for the method `alpha()` is provided in both super-types `TypeA` and `TypeB`.

If more than one super-type contains operations with identical names, there has to be a way to determine whether such situations lead to code selection ambiguities and, if so, eliminate them. An ambiguity occurs when re-definitions of a code implementation for operations from a super-type occur on several paths through the inheritance hierarchy. Different programming languages that support multiple code inheritance use different approaches to solve this problem. Some languages choose a particular super-type and qualify the ambiguous name with that super-type name. Other languages use renaming techniques.

Figure 2.1(a) illustrates the case in which different code for the method `alpha()` is provided in both super-types `TypeA` and `TypeB`.

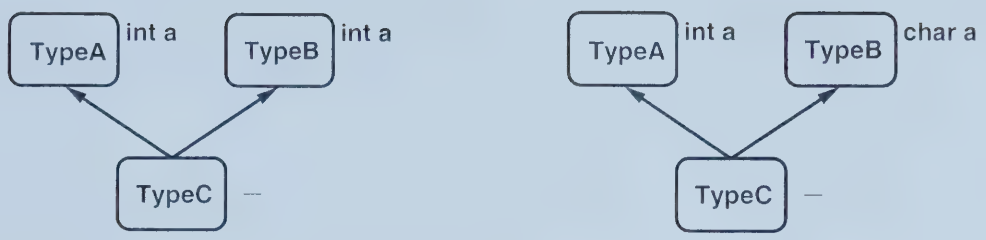
Since `TypeC` does not have an `alpha()` of its own (denoted by "–"), when `alpha()` is called on an object of dynamic type `TypeC`, a dilemma is encountered as to which implementation of `alpha()` should be inherited. In C++, the *use* of an ambiguous function generates a compiler error. To eliminate the

error, a programmer must provide code for the ambiguous method `alpha()` in `TypeC`. If the code in one of the super-types is wanted, the implementation of `alpha()` in `TypeC` can make a call to the appropriate super-type using a scope resolution operator (such as the `::` in the C++ approach), but a method that contains this call must be provided by the programmer.

Figure 2.1(b) shows a more complicated situation. An invocation of `alpha()` on an object whose dynamic type is `TypeC` may also be considered ambiguous since it could be argued that `TypeC` inherits code for `alpha()` indirectly from `TypeD` through two different paths, via `TypeA` and via `TypeB`. However, since the code is the same, there is no real ambiguity. C++ uses a modified *multiple sub-objects* approach for inheritance; multiple copies of a parent object can occur in the child object if, for example, the child inherits the parent indirectly on two different paths, as shown in Figure 2.1(b). Multiple sub-objects is the default, but in certain cases the programmer can specify that only one copy should be used. In C++, for the default inheritance case, this situation is considered an ambiguity.

Figure 2.1(c) shows an even more complicated situation. An invocation of `alpha()` on an object whose dynamic type is `TypeC` may also be considered ambiguous since it can be argued that `TypeC` inherits code for `alpha()` directly from `TypeA` and different code for `alpha()` indirectly from `TypeD` through `TypeB`. In C++, the compiler reports this as an ambiguity (for the default inheritance) and the programmer must define code for `alpha()` in `TypeC`. In Pang *et al.* [24] it is argued that, since the code for `alpha()` in `TypeD` is masked along *at least one path* by the code for `alpha()` in `TypeA`, there is not an ambiguity and the code from `TypeA` is inherited in `TypeC`. This less conservative definition of ambiguity is especially important if a language supports multi-dispatch [8] [9].

Our solution: For the situation in Figure 2.1(a), we mimic the C++ solution. For the situations in Figure 2.1(b) and 2.1(c), we can implement either the C++ solution or the less conservative version. Currently, we are using the less conservative definition of ambiguity, since we are also interested in Java multi-dispatch [4]. Because we do not yet have adequate compiler support for



(a) Data with the same type. (b) Data with different types.

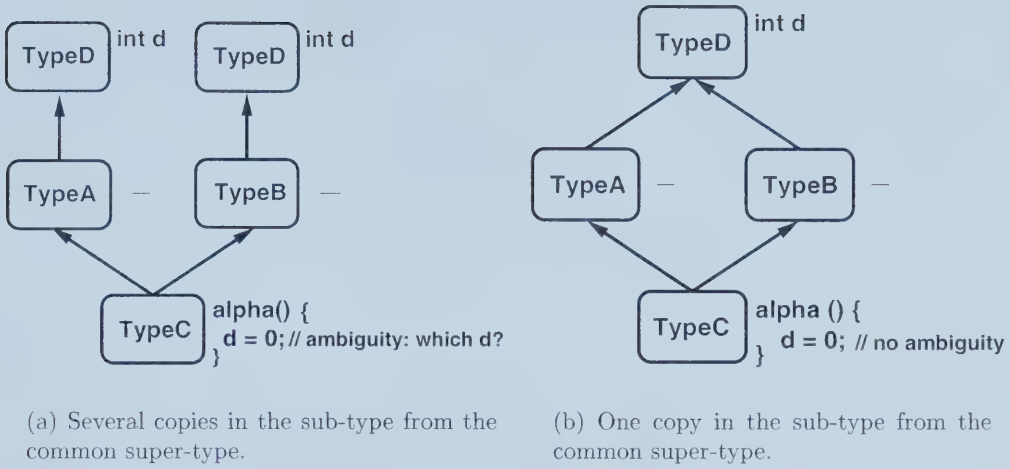
Figure 2.2: Data naming ambiguities: Case 1.

multiple code inheritance in Java, instead of signaling ambiguities at compile-time, we detect them at load-time (when the data structures associated with the sub-type are built) and, at that point, we throw an exception. If no ambiguities are detected, we proceed by executing the unambiguous method; the mechanism of choosing the method to invoke will be detailed in subsequent Chapters.

2.1.2 Problem 2: Data Naming Ambiguity

In languages with multiple inheritance, in addition to potential operation name clashes, data name clashes can also occur. Some languages maintain separate copies of data inherited from different super-types, while other languages merge like-named data together in the sub-type. If super-types contain common data, it has to be decided which copy of a data item coming from more than one path to use in a sub-type. For example, in Figure 2.2(a), if `TypeC` should only inherit one copy of the variable `a`, it does not matter if the copy "comes from `TypeA`" or "comes from `TypeB`", since they are both declared as `int`s. However, in Figure 2.2(b) it matters, since the variable `a` is an `int` in `TypeA` and it is a `char` in `TypeB`.

In C++, two uses of multiple data inheritance are distinguished with respect to the dependence relationship among super-types. First, if there are no dependencies among the super-types, then the object of the final sub-type must contain *sub-objects for each super-type*. Consider how inherited data item `a` is accessed in `TypeC` of Figure 2.2. Since there are sub-objects for each



(a) Several copies in the sub-type from the common super-type.

(b) One copy in the sub-type from the common super-type.

Figure 2.3: Data naming ambiguities: Case 2.

super-type, two copies of variable `a` are required in `TypeC`. Since there are two copies of `a`, when `a` is accessed in `TypeC`, an ambiguity occurs. As illustrated in Figure 2.2(b), data items may have identical names regardless of their types. C++ resolves both cases by using the scope resolution operator `::` (`TypeA::a` represents the `int a` in the `TypeA` part of the `TypeC` object and `TypeB::a` represents the `char a` in the `TypeB` part of the `TypeC` object).

Second, if there are dependencies among the super-types (two or more inherited types share a common type), the programmer has a choice. This kind of inheritance is also called *repeated inheritance*. By default, even if there is a common super-type (`TypeD` in Figure 2.3(a)) in the hierarchy, the sub-type (`TypeC`) will also contain *several* (two, in this example) sub-objects of that common super-type. Consider the case in which `TypeD` contains a data `int d`. `TypeA`, `TypeB`, and `TypeD` are normal C++ classes with the usual inheritance relationship, and consequently there are two copies of `int d` in `TypeC`, one inherited from `TypeD` via `TypeA` and the other inherited from `TypeD` via `TypeB`. If a method `alpha()` in `TypeC` accesses `TypeD`'s data item `d`, (for example, `alpha(){d=0;}`), then an ambiguity arises; it is not clear which of the two copies of `int d` in `TypeC` to use, the one inherited via `TypeA` or the one inherited via `TypeB`. C++ uses the scope resolution operator `::` to pick

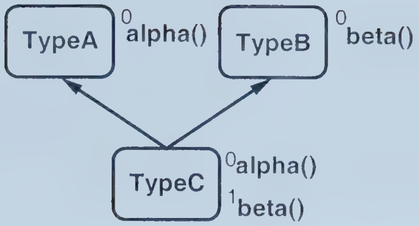


Figure 2.4: Code layout ambiguity.

one.

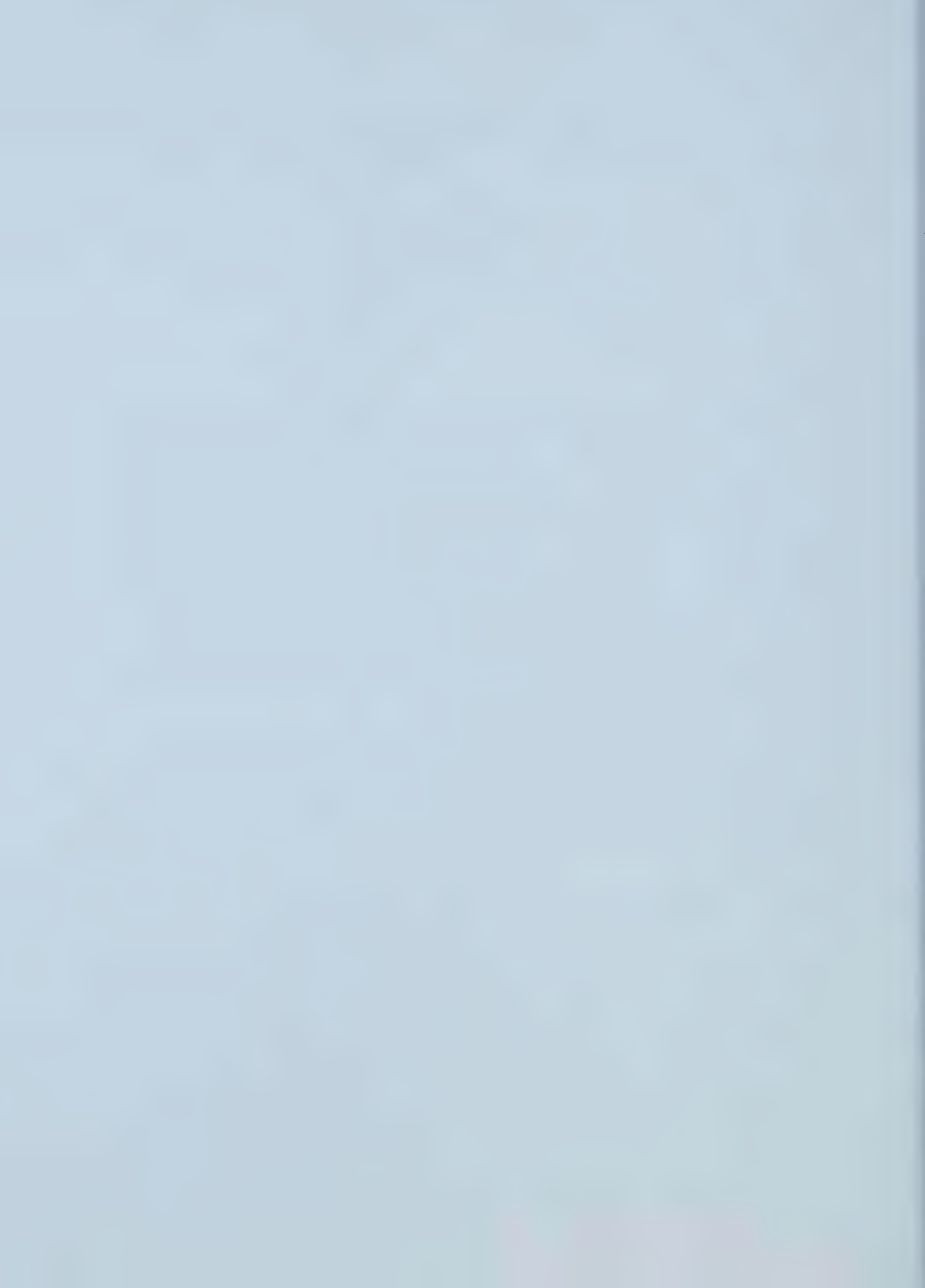
Alternately, the C++ programmer can specify that only one object of the common super-type resides in the final sub-type, the same object being shared in all sub-types. The C++ solution to resolving ambiguities is the following: if the derived class, `TypeC` in Figure 2.3(b), has to inherit *only one* copy of the data from the common class, `TypeD`, then the intermediate classes, `TypeA` and `TypeB`, need to declare the inheritance as *virtual*. Hence, there is just one copy of `int d` in `TypeC`, so accessing variable `d` does not generate an ambiguity.

Our solution: This problem does not exist in our implementation because we do not support multiple data inheritance.

The next two problems relate to multiple implementation (code and data) inheritance interaction. They are purely compiler issues regarding the layout of code and data, so they are not visible to the user. However, these problems must be resolved.

2.1.3 Problem 3: Code Layout Ambiguity

In the example from Figure 2.4, a problem arises from the different layout of the code in the sub-type (`TypeC`), with respect to the layout of the same code in the super-types (`TypeA` and `TypeB`). In the single inheritance situation, the same offset (i.e., 0) can be used to access the code for an operation in a sub-type and in its (direct or indirect) super-type. This is not the case with multiple code inheritance. When a sub-type inherits operations from several super-types, there is a problem in trying to set an order on the operations in the sub-type. In the example from Figure 2.4, should `alpha()` be placed before



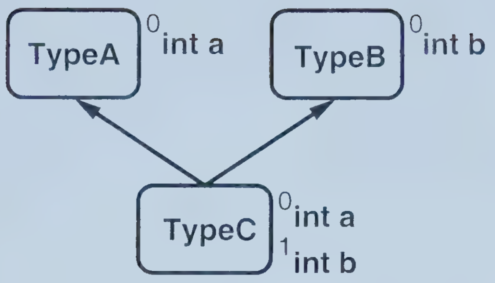
beta() in TypeC's data structures or after beta()? Regardless of our choice, we still have different offsets for one of the operations (beta() in Figure 2.4) in the super-type (offset 0 in TypeB) as compared to the sub-type (offset 1 in TypeC). This makes the single inheritance constant-index approach impossible for multiple inheritance.

Whenever we access methods of either TypeA, TypeB, or TypeC, the compiler must compute the offset of each method in the type's method table (*virtual function table* in C++). At run-time, this offset is used to access the appropriate method in the method table of the dynamic type of the receiver, even though the dynamic type of the receiver is not known at compile-time. For example, assume the method table in TypeC has the methods from TypeA, followed by the methods from TypeB, followed by any methods declared in TypeC as shown in Figure 2.4. The compiler can insert an offset of 0 into the code at a call-site for alpha(). At run-time, this offset can be used to access the code for alpha() in TypeA or TypeC depending on the dynamic type of the receiver. However, at a call-site for beta(), the compiler must select an offset of 0 to match the method table in TypeB or an offset of 1 to match the method table in TypeC. The solution in C++ is to use the offset of the super-type, but add a constant delta to the method table origin before adding the offset. The delta must be computed at run-time (delta $= 0$ for TypeB and delta $= 1$ for TypeC), since its value depends on the layout of the super-type and sub-type.

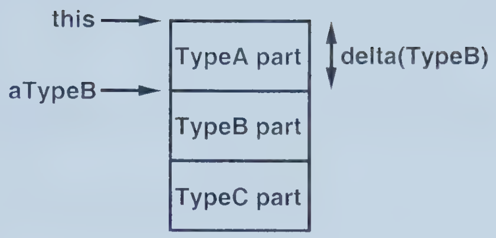
Our solution: Our approach is based on *interface method tables* that already exist in Java. In subsequent Chapters, we provide details about the interaction of data structures used to resolve multiple code inheritance in Java.

2.1.4 Problem 4: Data Layout Ambiguity

In the single inheritance case, data declared in a sub-type are concatenated with the duplicated data from the super-type in the sub-object image; therefore, a data item is located at the same offset in all objects of the super-type or sub-types. Since in the multiple inheritance case there is more than one super-type, a potential problem arises in establishing the layout of data in sub-type objects.



(a) Data layout ambiguity.



(b) The layout of a TypeC object.

Figure 2.5: Data layout issues.

The previous problem (Problem 3, Section 2.1.3) showed that method code could not be located at a fixed offset (table-index). A similar situation can occur for data. The offset of the data in the object image can change due to the same data being inherited from several possibly unrelated common parents. More importantly, in the case of multiple inheritance, the copies of the inherited data in the sub-object now have different offsets than the offsets that were known when the code which used them was compiled in the super-type. The situation in Figure 2.5(a) illustrates the case in which the super-types are not related. In C++, an object of `TypeA` contains an entry for each instance variable (data item). In our example, the only entry would be for the `int a`. Objects of sub-types (such as `TypeC`) are formed by concatenating the data of the super-type with their own data. In this case, a `TypeC` object would have two slots, one for `a` and one for `b`. It can be assumed that variable `b` follows variable `a` in `TypeC`'s object layout. For example, if we have a method `alpha(){b=0;}` in `TypeB`, when we compile it, we obtain the offset 0 for `aTypeB.b` (the offset of `b` in `TypeB`). When we invoke `aTypeC.alpha()`, the offset of `aTypeC.b` is 1 (the offset of `b` in `TypeC`), so it would be wrong to just use the compiled code for `alpha()` that uses the offset 0 even though `TypeC` is a sub-type of `TypeB`. This problem is more serious than the code layout ambiguity since each method compiled in a type (`TypeB`) that references instance variables can have the wrong data offsets, if it is applied to a sub-type (`TypeC`) object that inherits this code. The solution provided

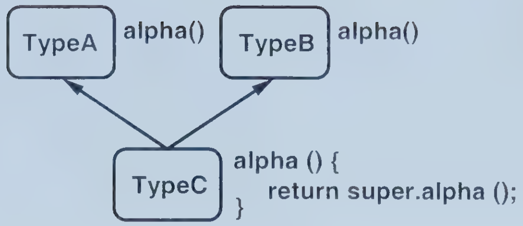


Figure 2.6: Super call ambiguity.

by C++ is the following: the *this* pointer (Figure 2.5(b)) which points to the start of the object layout is moved before the code is executed. For example, if the receiver object has `TypeC` as its dynamic type, the *this* pointer is moved to point to the start of the `TypeB` object when the method is called. An offset of 0 to access `b` now accesses the same `b` in a `TypeC` object, since the *this* pointer has been incremented by one word.

Our solution: Our approach to multiple code inheritance in Java does not support multiple data inheritance, so this problem is not applicable in our implementation of multiple code inheritance. Multiple data inheritance is a large source of problems and is not as useful as the code inheritance counterpart. Inheritance is beneficial when re-using code (more than it is for data), because the effort of programmers is mainly focused on implementing method bodies.

2.1.5 Problem 5: Super Call Ambiguity

The method code in a sub-type often *refines* the code of its super-types by adding some statements. In many object-oriented programming languages, this is accomplished by sending a message to the *super* object. Whenever a message is sent to *super*, the method lookup for that message starts in the super-type of the type that the method currently executing belongs to, instead of in the type of the receiver object. There are other approaches used to refine methods from the super-types, and some of them are shown in Table 2.1 of Chapter 2. However, *super* is the most popular refinement technique. When multiple inheritance is used, ambiguity problems with super calls may appear

due to the presence of multiple super-types. Figure 2.6 shows a refinement of alpha() in TypeC that contains an ambiguous super call. It is not clear whether the alpha() method in TypeA or TypeB should be called. Note that an ambiguous super call can exist even when no ambiguity occurs for the method that contains the super.

In C++, the super method call is qualified with the :: scope resolution operator. The lookup starts from the qualifying class.

Our solution: We extend the capability of the Java *super* keyword by specifying the superinterface from which the lookup for the given method starts. If code is found in the specified interface, then that code is executed. Otherwise, the superinterfaces are searched recursively. In the presence of ambiguities we throw an exception at load-time. If the lookup fails, we also throw an exception. We propose the syntax super(InterfaceA).alpha() for the future, which specifies the interface from which the lookup for method alpha() begins. Since our current implementation makes no language syntax changes, for now, we use a special marker in the source code just before the super call as described in Chapter 5.

2.2 Alternatives to Multiple Inheritance

Since Java does not provide *multiple code inheritance*, two idioms are commonly used to model complex applications that normally require this mechanism. Java libraries constitute a good source of examples in which these idioms are used in order to compensate for the lack of multiple code inheritance in Java. Figure 2.7 illustrates the hierarchical relationships among a few classes and interfaces from the java.io library.

2.2.1 Code Repetition

The simplest way to substitute for multiple inheritance is to *repeat the code* from the desired types into a sub-type, instead of simply inheriting it. The obvious drawback of this approach is an increase in code size. The hidden drawback is code deviation in which changes to a method are only made in

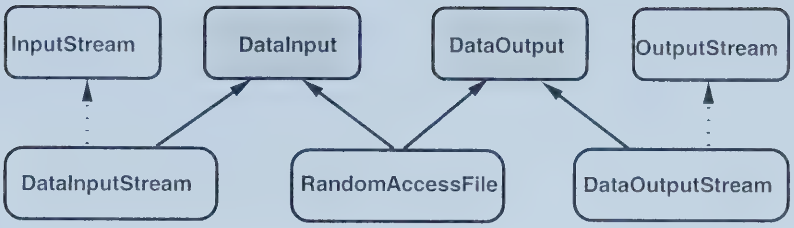


Figure 2.7: Some classes from the `java.io` library.

```
// java.io.DataInputStream and java.io.RandomAccessFile
public final float readFloat() throws IOException {
    return Float.intBitsToFloat(this.readInt());
}
```

Figure 2.8: Duplicate code in `java.io` library.

one copy, so that subtle bugs are introduced. Another disadvantage is that the source code must be available to the user for copying. Finally, the separation of inheritance concepts that we aim for is deteriorated, since both the interface-type and code-type levels are necessary for the user to be able to perform the required modifications.

The `java.io` library classes contain several examples of repeated code. One of them is the following: the classes `DataInputStream` and `DataOutputStream` implement the interfaces `DataInput` and `DataOutput` respectively. The class `RandomAccessFile` implements both `DataInput` and `DataOutput`, as illustrated in Figure 2.7. Much of the code that is in `RandomAccessFile` is identical or similar to the code in `DataInputStream` and `DataOutputStream`. As a specific example of identical code, consider the method `readFloat()` shown in Figure 2.8, which appears both in `DataInputStream` and `RandomAccessFile`. The methods `readFully(byte b[])` and `readDouble()` are also identical.

There are also many other similar methods, such as `readByte()`, shown in Figure 2.9, which differ only in the type of the receiver of some common methods such as `read()`. Other similar methods are the following: `readUnsignedByte()`, `readFully(byte b[], int off, int len)`, `readShort()`, `readUnsignedShort()`, `readChar()`, and `readInt()`. A num-

```
// java.io.DataInputStream and java.io.RandomAccessFile
public final byte readByte() throws IOException {
    int ch = this.in.read(); // int ch = this.read();
    if (ch < 0)
        throw new EOFException();
    return (byte)(ch);
}
```

Figure 2.9: Similar code in java.io library.

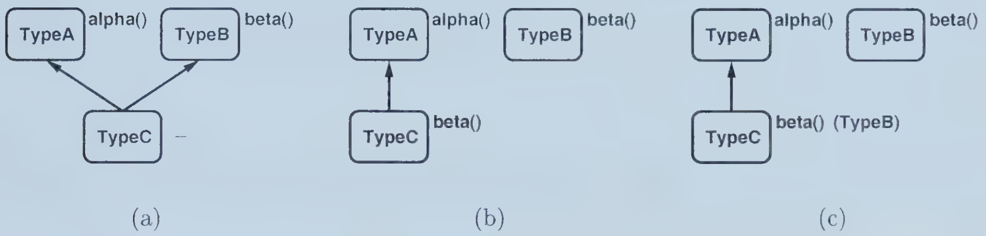


Figure 2.10: Delegation example.

ber of analogous identical methods can also be identified in the output stream classes `DataOutputStream` and `RandomAccessFile`, along with some similar methods that differ in the type of the receiver of some common methods such as `write(int)`.

2.2.2 Delegation

Delegation [35] allows an object to pass a received message to another object that is able to perform the task. This technique can be used in place of multiple code inheritance.

For example, the multiple-code inheritance in Figure 2.10(a) can be replaced by the single-code inheritance in Figure 2.10(b). In this case, each object of `TypeC` in Figure 2.10(b) has an instance variable that is bound to an object from `TypeB`. The method `beta()` is not inherited in `TypeC`. Instead, it has a one-statement implementation that invokes the `beta()` method in its sub-object of `TypeB`. We say that `TypeC` delegates `beta()` to `TypeB`. In general, the object that is delegated to may be stored as an instance variable or it may be passed as an extra method argument, as shown in Figure 2.10(c).

```
// class java.io.DataInputStream
...
public final int readInt() throws IOException {
    InputStream in = this.in;
    int ch1 = in.read();
    int ch2 = in.read();
    int ch3 = in.read();
    int ch4 = in.read();
    if ((ch1 | ch2 | ch3 | ch4) < 0)
        throw new EOFException();
    return ((ch1 << 24) + (ch2 << 16) + (ch3 << 8) +
        (ch4 << 0));
}
...
```

Figure 2.11: Example of delegation in `java.io.DataInputStream` class.

Unfortunately, this approach has the drawback of writing extra delegating methods and the overhead of sending more messages. In C++, it has been discovered that users found difficulties when designing based on delegation [31]. Overall, the burden is placed on the programmer to write extra code, preserve the return type and parameters list of the forwarding methods, and `throws` clauses whenever necessary. This supplementary work (writing methods that only delegate responsibility) is essentially done automatically when multiple inheritance is used.

The `java.io` library contains many examples of delegation. Figure 2.11 shows how class `DataInputStream` uses a reference (the instance variable `in`) to an `InputStream` to read characters that are assembled into an `int`. This is a simple example of delegation that is not used to replace multiple code inheritance.

A second example illustrates the way the `java.io` library copes with the absence of multiple code inheritance by using delegation. The class `File` (Figure 2.12) cannot simultaneously inherit from classes `ObjectInputStream` and `ObjectOutputStream`, since there is no multiple code inheritance in Java. However, in the implementation of `readObject()` and `writeObject()`, it needs the code of some methods from both classes `ObjectInputStream` and

```
// class java.io.File
...
private synchronized void
    writeObject(java.io.ObjectOutputStream s)
        throws IOException
    {
        s.defaultWriteObject();
        // Add the separator character
        s.writeChar(this.separatorChar);
    }
...
private synchronized void
    readObject(java.io.ObjectInputStream s)
        throws IOException, ClassNotFoundException
    {
        s.defaultReadObject();
        // read the previous separator char
        char sep = s.readChar();
        if (sep != separatorChar)
            this.path = this.path.replace(sep, separatorChar);
        this.path = fs.normalize(this.path);
        this.prefixLength = fs.prefixLength(this.path);
    }
...
```

Figure 2.12: Example of delegation in `java.io.File` class.

`ObjectOutputStream`. For this reason, an instance of one of these classes is passed as an argument to the method that uses their code and the read and write tasks are delegated to this argument.

An alternate approach to multiple inheritance, which ultimately results in delegation, is the use of *inner classes* inside interfaces [23]. However, this approach differentiates between using code from superclasses and superinterfaces, by using inheritance along the superclass chain and a form of delegation along the interface chains. For a class `ClassA` to use code from an interface `InterfaceA`, the programmer must explicitly declare a sub-object in `ClassA` and bind it to an instance of an inner class `ClassB` that extends an inner class `ClassC` declared in `InterfaceA`.

2.3 Advantages of Multiple Code Inheritance

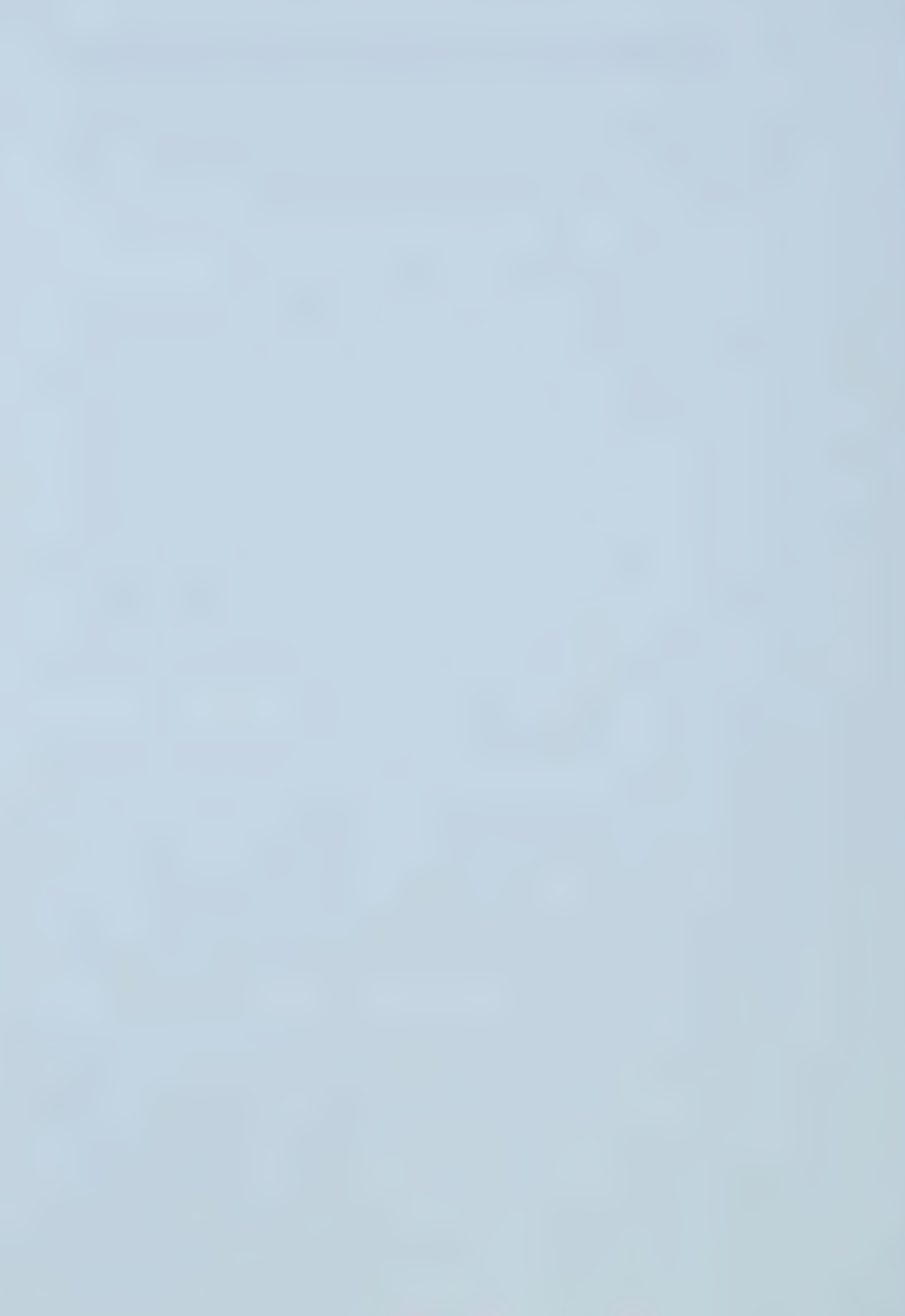
Our implementation of *multiple code inheritance* has the following **advantages**: facilitates code re-use, supports separation of inheritance concepts, and improves expressiveness and clarity of implementation.

2.3.1 Facilitates Code Re-use

Code re-use is manifested through code decrease due to increased code sharing. Multiple code inheritance can re-establish a certain degree of normality in the implementation of several Java applications. Commonality in the description of classes (method signatures) exists in Java and we can promote those features to common parent interfaces. Since Java has multiple inheritance of interfaces, it does not suffer from modeling problems. For example, the Java class `RandomAccessFile` implements the interfaces `DataInput` and `DataOutput`, as shown in Figure 2.7. Every instance of `RandomAccessFile` can be considered as both a `DataInput` and a `DataOutput`. This provides substitutability [14] so that any reference that is declared as a `DataInput` or `DataOutput` can be bound to a `RandomAccessFile`.

However, Java's lack of multiple code inheritance causes problems with implementation and maintenance. For example, even though `RandomAccessFile` implements `DataInput` and `DataOutput`, it cannot inherit code from these interfaces. Therefore, identical code appears in more than one class. For example, exact copies of the implementation of `readFloat` (Figure 2.8) appear in both `RandomAccessFile` and `DataInputStream`. This makes the program larger and harder to understand.

In addition, sometimes the code is incorrectly copied and often when changes are made to one copy, they are not made to all copies. In this example, because multiple code inheritance was not available, the Java library designers tried to simulate it by repeating and modifying the code where necessary in `DataInputStream`, `RandomAccessFile`, and `DataOutputStream`, increasing the overall code, instead of simply moving it up into the corresponding common super-types and subclassing accordingly. Thus, multiple code inher-



itance would result in a higher degree of code re-use; the programmer of a subclass no longer needs to be familiar with the specific implementation of the common operations.

Moreover, the re-use of code increases reliability, since it is common to find errors in repeated code when similar code is not consistent. An immediate consequence of code re-use is a decrease of maintenance costs.

Consider again Figure 2.9. To replace these methods by a common method, the line that differentiates them can be replaced by the common code: `int ch = this.source().read()`,
where `source()` is a new accessor method that for `DataInputStream` returns `this.in` and for `RandomAccessFile` returns `this`. The same abstraction can be used to share other similar methods, as shown in Section 2.2.1.

Although it would be possible to re-factor this hierarchy to make the class `RandomAccessFile` a subclass of either `DataInputStream` or `DataOutputStream`, it is not possible to make it a subclass of both, since Java does not support multiple-inheritance for classes. A re-factoring must accompany this abstraction, since the return type of the `source()` method must be specified as a single type that implements the operation, `read()`. The receiver of the `source()` method call in `DataInputStream` is referenced by the instance variable `in` that has static type `InputStream` (indirect superclass of `DataInputStream`). The receiver of the `source()` method call in `RandomAccessFile` is referenced by the pseudo-variable `this`, which has static type `RandomAccessFile`.

Unfortunately, in the current class/interface hierarchy, there is no common superinterface or superclass of `RandomAccessFile` and `InputStream` to use as the return type for the `source()` method. An interface must be added to the hierarchy that is a superinterface of `InputStream` and `RandomAccessFile` and declares the `read()` method.

However, after all of these common methods have been found, code inheritance has to be used to share them. Therefore, we need a common ancestor of `DataInputStream` and `RandomAccessFile` to store the similar `read` methods and a common ancestor class of `DataOutputStream` and `RandomAccessFile` to store the similar `write` methods. Since we are sharing code, this ancestor

should be a code-type.

The common code is ultimately factored into two code-types. A code-type implements the code for an interface and now a class implements the data for a code-type. Since there is no concept of code-type in Java, we must use either an *interface* or a *class* to represent our code-types.

2.3.2 Supports Separation of Inheritance Concepts

In addition to the increased degree of abstraction imposed by the clear separation among the three inheritance types, multiple code inheritance constitutes a necessary feature from a software engineering perspective "on the grounds that specification tools and implementation tools belong in a true software engineers toolkit." [30]. Programs can always benefit from having multiple views (designer, programmer, system administrator, user) of their design. Multiple code inheritance exploits code sharing to develop elegant and useful software components.

The separation of concepts emphasizes the role of interface-types, providing them with enhanced capabilities and control. Our multiple code inheritance approach does not allow code inheritance without interface inheritance. In this context, an aspect of major significance is the consistency of interface-types. In conjunction with *polymorphism* – a mechanism that supports inheritance, triggering several behaviors using the same interface – inheritance permits a super-type to define an interface-type for which several implementations are provided in the sub-types by means of code-types. When we lack information about sub-types, but we know the interface-type of the super-type, we can pass a reference to an object of the sub-type wherever a super-type is used. This way we can ensure that only behavior specified in the interface-type is called, the implementing types being hidden from the user.

2.3.3 Improves Expressiveness and Clarity of Implementation

Multiple inheritance supports a better organization of types, for the simple reason that it is congruent with real-world applications which make extensive

use of multiple features from unrelated concepts.

Since "class hierarchies can be used to organize and reason about software entities" [6] and since it is primarily an inheritance mechanism, multiple inheritance also extracts knowledge from the multiple type declarations and enriches types by providing them with more features.

Multiple code inheritance has the capability of enhancing expressiveness when implementing new systems, by thinking of a type as a sub-type of several other types. Since in our implementation we would like the code to be inherited from the interface methods, we would also like to have a subclassing relationship (code-type) which alone does not guarantee sub-typing. The blending of these two aspects leads to multiple specialization (*"is-a" relationship* [17] that we can find in the `java.io` library: `RandomAccessFile` is a specialization of both `DataInput` and `DataOutput`).

As multiple inheritance makes applications easier to design (via multiple interface inheritance) and implement (via multiple code inheritance), and equally easier to understand, it supports rapid prototyping and exploratory programming. Multiple inheritance reduces the time necessary to build and maintain applications. Applications are more comprehensible because the amount of new information is reduced. New types can be built taking advantage of existing ones, allowing for quick software development. The goal is to design software that is easy to use and modify – reusable software. We need to have the tools that help us build reusable software, and multiple code inheritance is a powerful tool.

2.4 Existing Multiple Code Inheritance Languages

We have investigated the mechanisms of multiple code inheritance in several programming languages in order to find out how common problems that occurred due to multiple code inheritance were solved. One of the issues of interest is *ambiguous name resolution*. When a class inherits the same operations/data (i.e., method signatures/instance variables) from multiple super-

types, we have a potential naming conflict. Another issue is the use of *super calls* when multiple code inheritance is present, because there are many supertypes to choose from. The procedure for the resolution of such ambiguous situations varies in each of the presented languages.

2.4.1 Ambiguous Name Resolution

There are several modalities to cope with inheritance conflicts and they can be grouped into *explicit* (disallowing conflicts, requiring the user to select a feature, or disambiguating with a resolution operator, such as `::` in *C++*) and *implicit* (choosing one feature by algorithmically resolving the conflict) resolution.

One category of programming languages demands the user to **explicitly** disambiguate name conflicts in the code. **Eiffel** [11] takes this approach. It has a `rename` clause that is used to solve name clashes. For implementation inheritance clashes, *Eiffel* combines its `rename` and `select` clauses to resolve ambiguities. **C++** delays this process until the ambiguous feature is **first used**.

In **Sather** [29] (originally based on *Eiffel*) the compiler enforces renaming of name conflicts. This is done *explicitly* by the user. The multiple inheritance is called "multiple inclusion".

In **Cecil** [3] all access to instance variables are through *accessor* methods. An object maintains space for each inherited copy-down variable, regardless of the names (distinct variables with the same name are not merged automatically). The problem reduces to resolving ambiguities among like-named accessor methods. Moreover, ambiguous variables could be accessed by a method in the child with the same name as an accessor method by means of directed re-send messages. Also, *Cecil* does not support repeated inheritance.

Another category of programming languages uses an **implicit** approach of resolving ambiguities.

Python [28] and **Perl** [25] follow a rule of pre-order traversal of the inheritance *tree* for both operation and data inheritance. The resolution rule employed is *depth-first, left-to-right.* Thus, if a feature is not found in a class,

its superclass tree is searched recursively upward depth-first. This approach, instead of the more intuitive *breadth-first* (searching all the immediate superclasses first and then their superclasses), helps decide if potential ambiguities occur. It accepts direct and inherited features of the first superclass before establishing if there are conflicts with the same features of a second superclass.

CLOS [5] performs a left-to-right linearization of its inheritance graph to a flat list. It further extends this approach to accommodate multi-dispatch by totally ordering multi-methods using argument positions, also ordering them from left to right.

Dylan [10] uses an *implicitly* performed linearization of the inheritance graph but it differs from *CLOS* because it does not take advantage of the argument positions when determining the method to execute.

Other programming languages that support multiple code inheritance simply discard any kind of naming conflicts. There have even been some cases where researchers have tried to add multiple inheritance to existing languages. For example, [1] attempts to add multiple inheritance to Modula-3 using mixins.

In **our implementation**, whenever a naming conflict is detected, a run-time exception is thrown when the class is loaded. With the proper future compiler modification, we can recognize these conflicts at compile-time instead of waiting until load-time.

2.4.2 Ambiguous Super Call Resolution

Sometimes, when using inheritance, we would like to be able to use in a type a corresponding method in one of its super-types, in order to enhance the functionality of the current method.

In **C++** (and **E** [18], designed as an extension to version 1.2 of *C++*) the user has to *explicitly* qualify names (class name followed by the scope resolution operator `::` and then the name of the method) to access methods from parent classes in the corresponding methods from the child classes. This technique starts the lookup in the specified class, rather than in the superclass of the current executing method.

In **Perl** the *super* mechanism performs a search through the object's inheritance tree, proceeding left-most, depth-first. This is triggered by the qualification of a method with the `SUPER` pseudo-class (a package is used as a class in Perl). However, the access `SUPER::` is only possible from inside the overridden method call.

Eiffel: Although it does not support the super call mechanism, it can create two versions of the *routine* (i.e., method in *Eiffel*) by inheriting the superclass twice, in one inheritance clause it uses `rename`, in the other it redefines the routine using `rename` and `select`.

Python: It combines the *"call-next-method"* pattern with the *method resolution order* (MRO given by the `__mro__` class attribute). A super call in *Python* has the following form: `super(className, self).alpha()`. The first argument to *super* is always the class in which the *super* occurs; the second argument is always `self`. The *super* expression searches `self.__class__.__mro__` (the MRO of the class that was used to create the instance in self) for the occurrence of `className`, and then starts looking for an implementation of method `alpha()` from that point on.

CLOS – *Implicit* linearization of the inheritance graph determines class precedence, which triggers method precedence. The inherited methods are linearized and the method to be executed is chosen using `call-next-method` *from the current method* in order to retrieve the next method in the chain. Furthermore, method qualifiers `before, after`, and `around` are used to communicate between the overriding method and the method in the superclass (the `inner` keyword plays an important role in this mechanism).

In **Sather** [32], the use of super calls is confusing in certain cases. The ambiguity arises when code that makes a super call is itself inherited. It is not clear if the inherited super call refers to the superclass of the original defining class A (A defines a `beta()` which contains `super.alpha()`) or of the inheriting class B (B extends A, so inherits `beta()`, but does not override it). Therefore, *Sather* replaces the super mechanism by *implicitly* renaming in the `include` clauses which define code inheritance. The `include` clause can be used to `include` and `rename` a single feature from another class or an entire

class. Renaming affects only the definition, not the calls of a specified feature.

Cecil: *Explicit* qualification. In *Cecil*, which has multiple dispatch, the qualification is based on multiple arguments.

Dylan: *Implicit* linearization of the inheritance graph determines class precedence, which triggers method precedence. The overriding method contains `call-next-method` in order to choose the right method from the list of method precedence implicitly built.

In **our implementation** for super calls, we have devised a qualification manner of lookup-start combined with a self-directed algorithm: the user has to provide the name of the type where the lookup will commence. From there up, if no code is found, the lookup is further controlled by an algorithm that unambiguously determines a super-type with code (if any) for the given method. We propose a syntax for the future, `super(Start).alpha()`, where the lookup starts in the interface type Start.

2.5 Concluding Remarks

In this Chapter, we continued to motivate the need for multiple inheritance started in Chapter 1. Therefore, we analyzed the state of multiple inheritance, beginning with the major problems associated with it, and we described our solution to each of them.

Then, we saw that the alternatives to multiple code inheritance had several drawbacks, including increasing the code size, introducing errors in programs when copying or modifying code, deteriorating the separation of inheritance concepts, writing extra delegating methods, and introducing the overhead of extra message sends.

Fortunately, we can avoid these problems by using multiple code inheritance, which has the advantage of facilitating code re-use, supporting separation of inheritance concepts, and improving the expressiveness and clarity of implementation.

We investigated the mechanisms of multiple code inheritance in several programming languages in order to find out how common problems that occurred

due to multiple code inheritance were solved.

Since Java has multiple inheritance of interfaces, but only single inheritance of code, our solution to the problems generated by this situation is to generalize a JVM to support multiple inheritance of code, by inserting code into interfaces. We support multiple code inheritance, not multiple data inheritance, because the latter is not as useful as code inheritance. Re-using code is more important, since the effort of programmers is mainly focused on implementing method bodies.

In later Chapters, we describe our JVM modifications to support multiple code inheritance in Java and propose future syntax for super calls to interfaces.

Language	Operations	Data Naming	Code Layout	Data Layout	Super Calls
C++, E	*Explicit*; signals ambiguities at **first use**.	*Explicit*; signals ambiguities at **first use**.	Multiple *vtbls* offsets adjust *this* pointer.	Also moving *this* pointer.	*Explicit*; qualification with starting lookup class.
Eiffel	*Explicit*; *rename* clause.	*Explicit*; *rename* and *select* clauses.	*Explicit*; *rename* clause.	*Explicit*; *rename* and *select* clauses.	No explicit super call mechanism.
Python	*Implicit*; pre-order traversal inheritance tree.	*Implicit*; pre-order traversal inheritance tree.	*Implicit*; pre-order traversal inheritance tree.	*Implicit*; pre-order traversal inheritance tree.	*Method resolution order* combined with *call-next-method*.
Perl	*Implicit*; pre-order traversal inheritance tree.	No data inheritance.	No data inheritance.	*Implicit*; pre-order traversal inheritance tree.	Depth-first, left-to-right resolution. Methods qualified with SUPER pseudo-class.
CLOS	*Implicit*; inheritance graph linearization.	*Implicit*; inheritance graph linearization. Merges members with the same name into a single slot.	*Implicit*; inheritance graph linearization, taking into account arguments' positions.	*Implicit*; inheritance graph linearization.	*Implicit*; inheritance graph linearization. Methods communicate by keywords: *before, after, around, inner.*
Sather	*Explicit*; compiler enforced conflict renaming.	*Explicit*; compiler enforced conflict renaming.	*Explicit*; compiler enforced conflict renaming.	*Explicit*; compiler enforced conflict renaming.	*Implicit*; renames features in *include* clauses.
Cecil	*Explicit*; qualification based on multiple arguments due to multiple dispatch.	Problem reduced to resolving like-named field accessor ambiguities.	*Explicit*; qualification based on multiple arguments due to multiple dispatch.	Problem reduced to resolving like-named field accessor ambiguities.	*Explicit* qualification based on multiple arguments.
Dylan	*Implicit*; inheritance graph linearization.	*Implicit*; inheritance graph linearization.	*Implicit*; inheritance graph linearization.	*Implicit*; inheritance graph linearization.	*Implicit*; graph linearization. Methods chosen with *call-next-method*.

Table 2.1: Programming languages that support multiple code inheritance.

Chapter 3

Method Dispatch in the JVM

Before describing the changes made to the Sun's JVM for JDK 1.2.2 implementation, we look at some key data structures for storing information and performing method dispatch in the original JVM [22]. Later on, we will discuss how the method dispatch in the JVM is modified.

3.1 Overview

The term Java is broadly used to indicate four technologies: the Java programming language [13], the Java `.class` file format, the Java Application Programming Interface (API), and the Java Virtual Machine (JVM). The Java API and the JVM form the *Java Platform* on which every Java program can run, regardless of the underlying hardware or operating system of the platform. The philosophy of Java programs is "write once, run everywhere". The JVM implementation described in this dissertation is Sun JVM for JDK 1.2.2 [26] for Linux.

A Java program is compiled into a sequence of bytecodes and the JVM, an abstract computer able to run Java programs, interprets the bytecodes. Each class or interface is compiled (with `javac` or another compiler producing bytecodes) into a binary format `.class` file. When the JVM loads a `.class` file, it parses the information about the class or interface from the binary data and places it into run-time data structures within the method area. Then, it executes the bytecodes from the `.class` file. Along with the program's `.class` file, the class loader also loads the necessary classes from Java API. The JVM

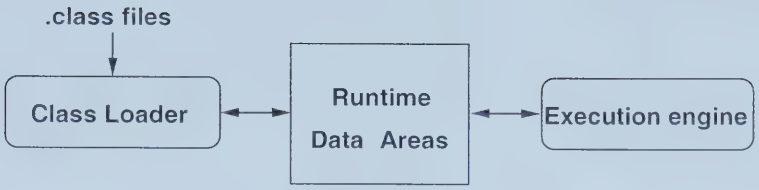


Figure 3.1: Simplified JVM internal architecture.

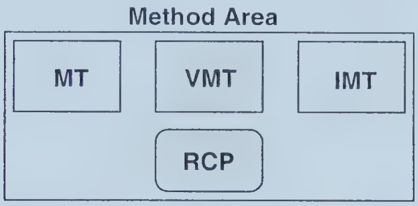


Figure 3.2: Method Area within the Runtime Data Areas: Method Table (MT), Virtual Method Table (VMT), Interface Method Table (IMT), and Runtime Constant Pool (RCP).

accomplishes these two tasks through the class loader [34] and the execution engine (Figure 3.1).

A `.class` file stores all of its symbolic references to other types needed by the current class in its **constant pool** (CP), which is a sequence of constant items with a unique index. These items can be *literals* (strings, integers, floating point constants) or *symbolic references* to types (classes and interfaces), fields, and methods that have to be determined at run-time. In addition to the constant pool, which represents the information referenced from the current class, a `.class` file also contains information about the fields and methods declared in that type: a field information structure (for each field's name and type) and a method information structure (for each method's name and descriptor, bytecodes and other information). Conceptually, the CP is similar to the symbol table of other programming languages and systems.

Once loaded by the JVM, a type has an internal version of its constant pool in the form of a **runtime constant pool** (RCP) that is stored in the method area as shown in Figure 3.2. All of its symbolic references now reside in the type's runtime constant pool. Instructions refer to CP indexes where

```
FileInputStream aFile = new FileInputStream("aFile.txt");
FilterInputStream stream;
stream = new DataInputStream(aFile);
stream.close();
```

Figure 3.3: Java sample source file.

the symbolic references reside. During the running of a program, if a symbolic reference has to be used, it must be *resolved* (i.e., replaced with a direct reference). *Dynamic linking* is the process of *locating* types (classes and interfaces), fields, and methods referred to by symbolic references stored in the constant pool and *replacing* them with direct references to data. The constant pool has a central role in the dynamic linking of Java programs. *Direct references* to types (class/interfaces), class variables, class methods are represented by **pointers** to data in the method area. *Direct references* to instance variables and instance methods are represented by **offsets**. Instance variables are offsets from the start of the object's image to the location of the instance variable, and instance methods are offsets into the virtual method table (array of pointers to methods data in the method area).

Consider the call-site `stream.close()` from Figure 3.3. In the method information section in the `.class` file for this call-site, the bytecodes which use the constant pool indexes are illustrated in Figure 3.4. In the first part of this Figure there are the raw bytecodes (in hexadecimal format) followed by a comment with their "translation" into JVM instructions mnemonics. The second part of the same Figure contains only the instructions automatically generated with the `javap .class` file disassembler tool. We will trace this example in the next Section, which presents the way the JVM uses the `.class` file information. The information from the constant pool used by this call-site is represented in Figure 3.5. Recall that the CP is an array.

Entries in the constant pool begin with a tag which indicates the kind of constant stored. For example, if the entry is a class (entry `5`, for example), then its tag is `CONSTANT_Class`; if the entry is a method (entry `11`), its tag is `CONSTANT_Methodref` (respectively `CONSTANT_InterfaceMethodref` for inter-

```
// Snippet from the .class file (output in hexadecimal format)
// for the stream.close() call-site.
2c //aload_2
b6 //invokevirtual
000b // #11 java.io.FileInputStream/close

// Snippet from javap output for the same call site.
19 aload_2
20 invokevirtual #11 <Method void close()>
```

Figure 3.4: Snippet of the .class file.

```
...
5 Class #31
...
11 Methodref #5 #15
...
15 NameAndType #28 #16
16 Utf8 "()V"
...
28 Utf8 "close"
...
31 Utf8 "java.io.FileInputStream"
...
```

Figure 3.5: Snippet of the constant pool.

face methods). For brevity, we do not use `CONSTANT` in front of CP tags in our examples.

In order to actively use a type, the following steps are taken:

1. Loading: responsible for importing a binary form for a type into the JVM (*generating* a stream of binary data representing the type out of the fully qualified type name, *parsing* this stream into internal data structures in the method area, and *creating* the type as an instance of class `java.lang.Class` on the heap). All of the type's super-types (classes and interfaces) have to be loaded before loading the actual type.

2. Linking: responsible for the integration of the binary data into the runtime structures of the JVM

   (a) Verification: validates the loaded type.

   (b) Preparation: allocates memory for the class variables and sets them to default initial values determined by their types. It also allocates memory for data structures such as the **method tables** (MT), the **virtual method tables** (VMT), and the **interface method tables** (IMT).

   (c) Resolution (optional step): replaces symbolic references into the constant pool with direct references to data. This step is delayed until each symbolic reference is first used by the program.

3. Initialization: responsible for providing the class variables with their proper initial values. For classes, the class's direct superclasses have to be initialized first if they have not already been initialized. If there is a class initialization method (`<clinit>`), it will be executed. This is a special method created by the Java compiler and contains all the class variable initializers and static initializers of a type; it can be invoked only by the JVM. Final static variables are not stored as class variables in the method area but as constants into the constant pool.

3.2 Method Invocation Mechanism

In Java, there are two categories of methods that can be invoked: **instance methods** – the JVM selects the method to invoke based on the actual class of the receiver object (run-time, dynamic binding) and **class (static) methods** – there is no receiver object so the method is actually a function defined in a class (compile-time, static binding).

There are four invoke instructions in the Java `.class` files: `invokevirtual`, `invokeinterface`, `invokespecial`, and `invokestatic`.

The instruction executed at a call-site (for example `stream.close()`) depends on the static type of the receiver (`stream`). This call-site is translated into a JVM instruction whose opcode is `invokevirtual` if the static type of `stream` is a *class* and `invokeinterface` if the static type of `stream` is an *inter-*

face. In both situations, the opcode is followed by a *method reference* (an index into the constant pool) as an operand of the instruction. The method reference stores the static type of `stream` (class or interface) and the method signature of `close()` (name and descriptor – the return type and arguments). In Figure 3.4 and Figure 3.5, the index is `11` and it indicates a `Methodref` tag into the CP. The entry `Methodref` has two fields which point to other structures in the constant pool: *class* and *descriptor*. In our example, the `Methodref` tag points to entries `5` (`Class` tag) and `15` (`NameAndType` tag) in the CP. The `Class` tag indicates that at entry `31` the string with the fully qualified class name (where the method is defined) can be found. The `NameAndType` tag indicates the CP entries where the name of the method (`28`) and the signature (`16`) are found. Therefore, given an invoke instruction and an index into the CP, the name and type of the method, as well as its static class are retrieved. In this particular example, given the call-site, the index following the invoke instruction provides all the information necessary for the method signature to be statically determined. The receiver object, though, is necessary to uniquely determine the method to be executed.

References to methods are initially symbolic: they refer to constant pool entries that contain symbolic references. When the JVM encounters an `invoke` instruction, it resolves it (if not yet resolved) as part of its execution. To **resolve a symbolic reference** to a method, the JVM **locates** the method being referred to symbolically (method lookup) and **replaces** the *symbolic reference* with a *direct reference* (pointer or offset). The class and name (including the signature) of the method are resolved before the method is actually invoked and an index into the virtual method table of the static type of the object is generated. Therefore, in future invocations, the JVM will be able to execute methods more quickly, as we will see in the next Chapters.

When invoking a Java non-native method, the JVM creates a new stack frame for each Java method it invokes and pushes the stack frame onto the Java stack. The new stack frame contains local variables of the method, the operand stack, as well as other implementation-dependent information. For every instance method invocation, the JVM expects a reference to the object

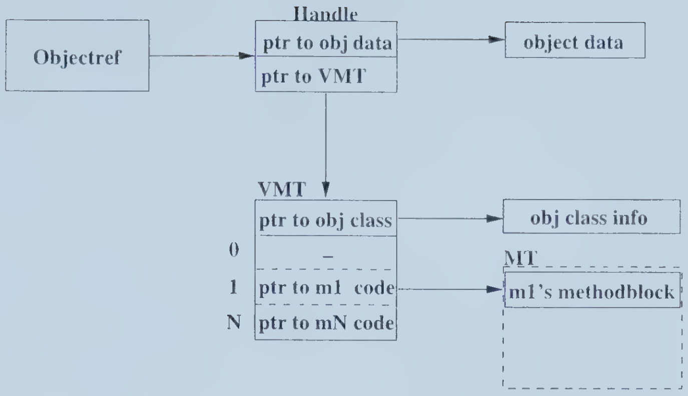


Figure 3.6: High-level object representation in Sun's JVM.

(we will refer to it as *objectref*, i.e., the implicit *this* pointer that is passed to any instance method, representing the receiver object), as well as the arguments (if any) required by the method to be on the operand stack of the calling method's stack frame (class methods require only the arguments). They must be pushed onto the calling method's operand stack by the instructions that precede the invoke instruction. The JVM places them as locals on the new stack frame. The JVM makes the new stack frame current and sets the program counter to point to the first instruction in the new method.

3.3 Object Representation

In Sun's JVM, each object reference (*objectref*) is a pointer to a structure which contains a pointer to `methodtable` (VMT) and a pointer to the object's instance data (Figure 3.6). The VMT has a pointer to the full class data and an array of pointers to method data containing the actual information for each instance method that can be invoked on objects of that class. The method data (structure named `methodblock` in SUN's JVM) pointed to from a slot (entry) of the virtual method table (or of the interface method table via the VMT) contains the compiled code for that method, i.e., complete information about the method. A `methodblock` includes the size of the operand stack and local variable sections of the method's stack, a pointer to the method's

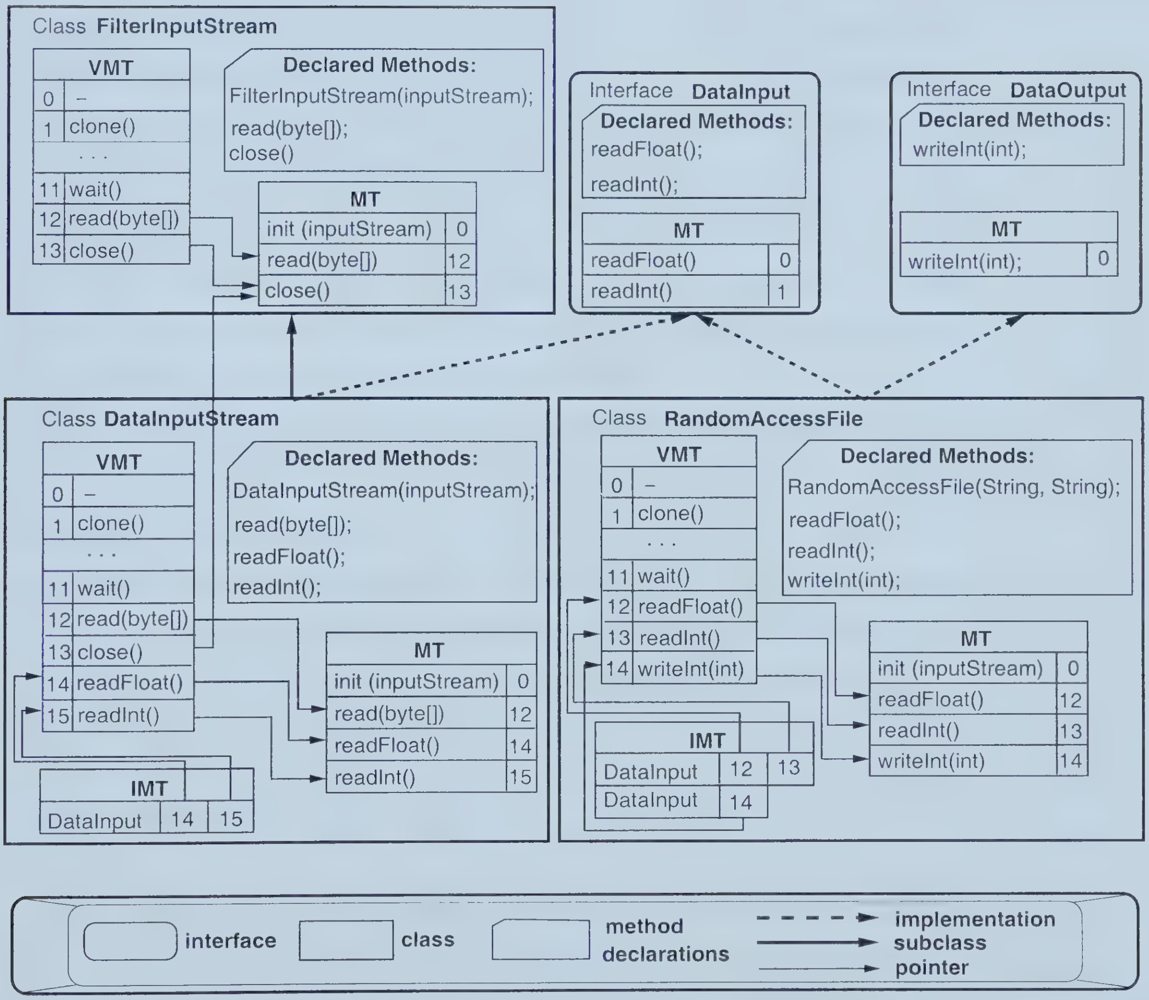


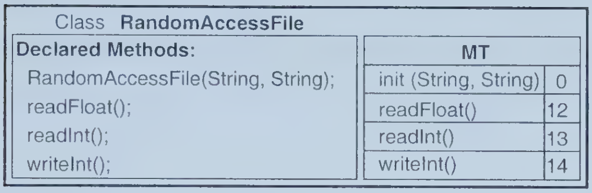
Figure 3.7: `DataInputOutput` example: The MT, VMT, and IMT for some classes and interfaces from the `java.io` package.

bytecodes, the method signature, and an exception table. The methodblocks are organized into an array in the method table (MT). The virtual method table (VMT) includes pointers to data for methods declared explicitly in the object's class or inherited from superclasses. For interfaces, the code pointer is currently *null*, but we change this as described in Chapter 4. Having only a reference to an object (*objectref*), we will see in subsequent Chapters how we can retrieve information about that object's class. The methodblock to be executed depends on the runtime type of the receiver object, therefore first the *objectref* is located by popping all the arguments from the stack. Then the object handle is retrieved. The handle is used to locate the VMT of the actual class of the object and, given the index (Figure 3.6) into VMT that was generated (as the index is identical in all the VMT of classes which implement that method), the desired methodblock is fetched.

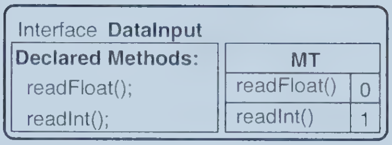
We exploit the existing structure of the original JVM. There are three data structures that contain method information: the **method table** (MT), the **virtual method table** (VMT), and the **interface method table** (IMT). Every class and interface has an MT. Every class has a VMT, but interfaces do not have VMTs, because interfaces are never instantiated. Every interface and every class that implements an interface (directly or indirectly) has an IMT.

3.4 The Method Table

An MT is an array of methodblocks, one for every method that is declared (**not** inherited) in a class or interface. Therefore, the **method table** contains methodblocks for all overriding methods. In Sun's JVM, an MT is a data structure called methods. Each methodblock contains all of the information about the method, including its signature and a pointer to its bytecodes. In the case of interfaces, the methods are abstract, so the code pointer is not used (but we will modify the JVM to use it, as described in Chapter 4). Figure 3.7 shows the classes FilterInputStream, DataInputStream, RandomAccessFile, and the interfaces DataInput and DataOutput from the



(a) MT for Classes.



(b) MT for Interfaces.

Figure 3.8: Method tables.

`java.io` package. Many methods have been excluded for the sake of simplicity.

Each of the classes has one `methodblock` in its MT for each declared method in the class. For example, `DataInputStream` has a `methodblock` for `read(byte[])` since it overrides this method that it inherits from the class `FilterInputStream`, but has no `methodblock` for `close()` since this method is not overriden. The interface `DataInput` has `methodblocks` for its methods `readFloat()` and `readInt()`, even though they contain no code.

Method dispatch finds a `methodblock` for a call-site and invokes the code for the `methodblock`. The operand of the call-site bytecodes is an index into a run-time constant pool that stores the signature of the method being invoked. Method dispatch is a two-step process. The first part of method dispatch, called *resolution*, finds a `methodblock` that contains the code. The resolution mechanism depends on whether the static type of the receiver object is a class or an interface. The compiler records the required resolution mechanism in the `.class` file by generating an `invokevirtual` bytecode if the static type of the receiver object is a class and an `invokeinterface` bytecode if the static type is an interface. Resolution of `invokeinterface` is complex and is discussed later in this Chapter. Resolution for `invokevirtual` is simple.

To resolve an `invokevirtual` instruction, the JVM uses the method reference to obtain the static class and a method signature. It then searches the MT of this class for a `methodblock` with a matching signature. If no match is found, it searches the MTs along the superclass chain. The compiler guarantees that a match is found. Consider Figure 3.9: the call-site `stream.close()` has bytecodes that contain an index into the run-time constant pool that stores

[illegible]

the method signature, `close()`. The dynamic class of the receiver object may be `DataInputStream` or `FilterInputStream`. If it is the former, the MT table of `DataInputStream` is searched for a `methodblock` with signature `close()`, but no match is found. The MT of the superclass, `FilteredInputStream`, is searched, and a `methodblock` with a matching signature is found.

However, it is possible that the *resolution* `methodblock` is not the correct *execution* `methodblock`. For example, consider the classes in Figure 3.7, a variable declaration, `FilterInputStream input`, and the following call-site: `input.read(aByteArray)`, where `input` is bound to an instance of the class `DataInputStream`. Resolution produces the *resolution* `methodblock` for `read(byte[])` in class `FilterInputStream`. The *execution* `methodblock`, however, should be `read(byte[])` in `DataInputStream`. If the index of a `methodblock` in its MT were invariant along the superclass chain, the resolved `methodblock` `read(byte[])` in `FilterInputStream` could store this invariant index, and it could be used as an index into the MT of the dynamic class of the receiver object (`DataInputStream` in this example). Unfortunately, MT indexes are not invariant. Fortunately, this problem is solved using virtual method tables as described later in this Chapter. In essence, each method block contains a unique VMT index that is invariant along the superclass chain.

Resolution is quite slow, so Sun's JVM records the resolution result at each call-site for use in future dispatch at that same call-site. *Bytecode quicking* (described in more detail later in this Chapter), modifies the bytecodes at each `invokevirtual` call-site to contain information that can be used to quickly compute an index into a VMT that contains a pointer to the appropriate `methodblock`. More specifically, the bytecodes will contain a reference to the resolved `methodblock` instead of the original symbolic method reference.

3.5 The Virtual Method Table

The **VMT** enables the JVM to quickly locate an instance method invoked on an object. In Sun's JVM, this data structure is called `methodtable` (not to

```
...
FilterInputStream stream;
DataInput input;
RandomAccessFile file;
...
if (condition)
    stream = new FilterInputStream(inStream);
else
    stream = new DataInputStream(inStream);
...
stream.read(anArray);
value = input.readint();
...
stream.close();
file.writeInt(43);
```

Figure 3.9: Code example.

be confused with the MT). The **virtual method table** is a data structure used to store invariant indexes for all methods along a subclass chain. Each VMT entry (*slot*) holds a reference to an instance method (i.e., a method that may be invoked on a class instance) implementation that has been *declared* or *inherited* by the current class. Each reference is actually a pointer to a `methodblock` in either the local MT or an MT of a superclass. The first entry in VMT (at index 0) is not used in this JVM implementation.

The MT and VMT for a class are constructed when the class is loaded, and each `methodblock` in the MT stores its VMT index as it is built at load-time. A VMT is similar to a virtual function table used in C++ implementations, except that in C++ the compiler inserts a virtual function table index at the call-site of each virtual function call. In Java, the compiler inserts a symbolic reference to the method at each call-site, and the first execution of the call-site resolves this symbolic reference and modifies the bytecodes at the call-site so that future executions use an index into the VMT. The VMTs contain only non-private instance methods. Private methods and instance initialization methods do not appear in VMTs because they are statically (i.e., compile-time) bound. The same is true of static methods.

Consider the classes `FilterInputStream` and `DataInputStream` in Figure

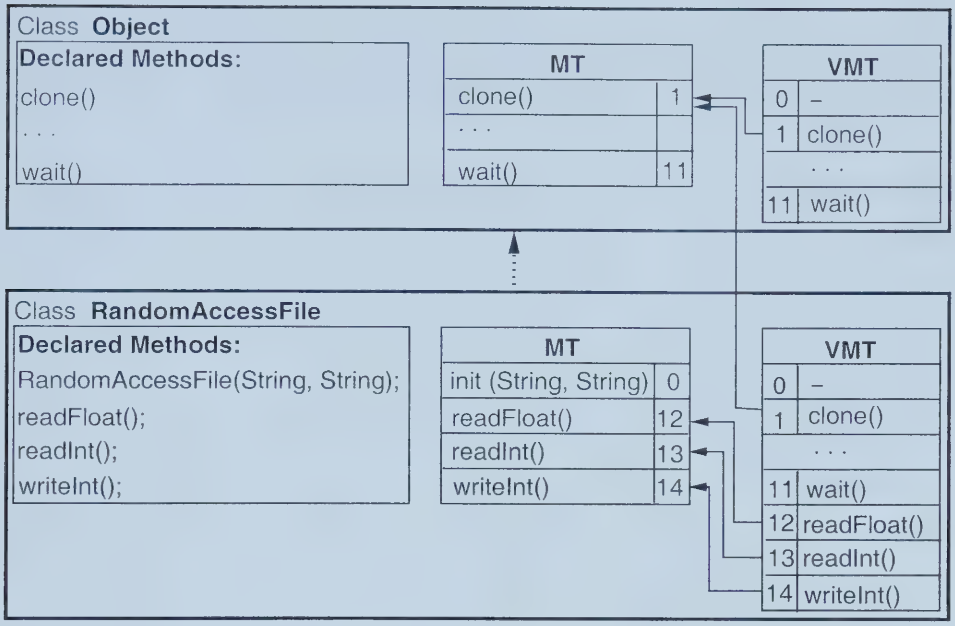


Figure 3.10: Virtual Method Tables.

3.7. The VMT of a class has indexes for all the methods it inherits from class `java.lang.Object` (indexes 1-11 in all VMTs in Figure 3.7) and then indexes for all of its other ancestor classes, ending with its immediate superclass (indexes 12 and 13 in the `DataInputStream` VMT). If a child class overrides an inherited method, it actually overwrites the VMT entry of the inherited method to refer to an entry in the local MT table rather than the MT table of an ancestor class. For example, the VMT entry for the overriding `read(byte[])` method in `DataInputStream` points to the local MT table. Finally, the VMT has indexes for all new methods that it declares (indexes 14 and 15 in `DataInputStream` VMT), even if these new methods implement methods from an interface.

Even though a child class overwrites a VMT entry to point to a `methodblock` in a different MT, a `methodblock`'s VMT index does not vary along a superclass chain. This is because when a VMT is constructed, it first copies its superclasses' VMT and then extends it. For example, the indexes of all methods inherited from `java.lang.Object` are the same in all VMTs and the indexes of

```
executeMB = receiver.dynamicClass.VMT[resolvedMB.vmtIndex]
```

Figure 3.11: Computing the execution `methodblock` for `invokevirtual`.

the `close()` and `read(byte[])` methods are the same in `FilterInputStream` and `DataInputStream`. This property is essential to support substitutability [14] for fast method dispatch, after bytecode quicking. For example, consider the code in Figure 3.9. At the call-site, `stream.read(anArray)`, the bytecodes initially contain a reference to a constant pool entry for `read(bytes[])`. If the dynamic type of stream is `FilterInputStream` when the call-site is encountered the first time, method resolution will generate the VMT index 12 and bytecodes will be quicked to use this index the next time the call-site is executed. If the stream variable is bound to a `DataInputStream`, the second time the call-site for `stream.read(byte[])` is executed, the same index, 12, will be used to access the same `methodblock`, but this time via the VMT for class `DataInputStream`. After resolution, the *execution* `methodblock` is computed from the *resolution* `methodblock`, `resolvedMb`, using the formula in Figure 3.11.

Unfortunately, this dispatch mechanism does not work for interfaces due to multiple inheritance. Figure 3.7 illustrates the problem, showing that a method `readInt()` declared in an interface `DataInput` that is implemented by two classes, `DataInputStream` and `RandomAccessFile`, can have different indexes (15 and 13) in the VMTs of the two classes. This can occur because each of the classes may inherit methods from different superclasses or implement different interfaces. Therefore, we need another data structure, **interface method table** (IMT), which facilitates interface method dispatch (i.e., `invokeinterface`).

3.6 The Interface Method Table

An IMT is used for interface method dispatch (`invokeinterface`). The corresponding data structure in Sun's JVM is `imethodtable`. If a variable has a static type that is an interface and if it appears as the receiver of a method

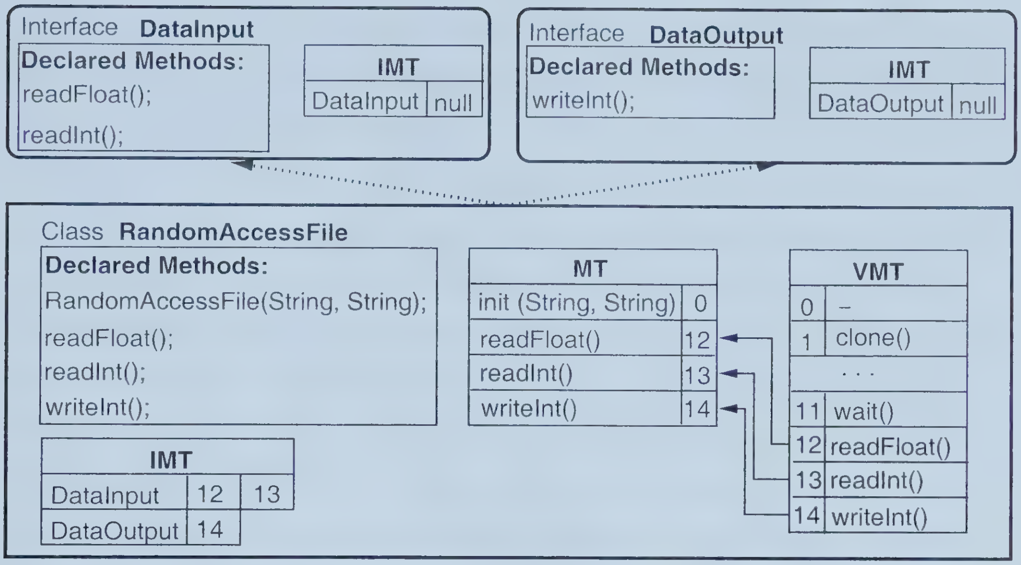


Figure 3.12: Interface Method Tables.

invocation, the call-site will contain an `invokeinterface` bytecode instead of an `invokevirtual` bytecode. We shall see how the IMT provides an extra level of indirection that solves the problem of inconsistent indexing of interface methods among classes.

Each slot in an IMT stores all of the information for an interface. Every *class* has an IMT that references all of the interfaces it implements or inherits. For example, in Figure 3.12, the class `RandomAccessFile` has two entries in its IMT, one for the interface `DataInput` and one for the interface `DataOutput`. Each *interface* also has an IMT with slots for all the interfaces it extends, including itself.

During method dispatch, the MTs of all of the interfaces that are implemented by the receiver object's class can be accessed through the IMT for that class. The IMT is an array of entries which contain two types of information. Each entry is a pointer to the interface that the class implements (directly or indirectly). Each entry also contains an array of indexes into the class's VMT; the number of elements in each array is the same as the number of methods that are in the interface referenced by the interface pointer of that entry. Each index is an offset into the VMT entry for the corresponding

method. For example, consider the IMT for the class RandomAccessFile in Figure 3.12. The first entry contains a pointer to DataInput and an array containing indexes (12 and 13) into the VMT for the two methods declared in DataInput, called readFloat() and readInt(). The second entry contains a pointer to DataOutput and an array containing an index (14) into the VMT for the method declared in DataOutput, called writeInt(int).

Resolution of an invokeinterface bytecode is similar to resolution for an invokevirtual bytecode, except that the method reference includes an *interface* instead of a *class*. Resolution starts at the **interface method table** (IMT) of this interface. Recall that an IMT has one entry for each interface that is extended or implemented (directly or indirectly) by its class or interface. During resolution, the JVM starts with the entry zero of the interface's IMT, which is the interface itself. The MT of this interface is searched for a matching method. If one is not found, the MTs of subsequent interfaces in the IMT are searched. The compiler guarantees that a signature match will be found.

Also, recall that the *resolution* methodblock may not be the *execution* methodblock. In the invokevirtual case, the resolved methodblock contains an invariant index into the VMT of the receiver object's dynamic class. In the invokeinterface case, the *resolution* methodblock contains a local MT offset. To complete the dispatch, an index must also be computed. The index is for the IMT of the dynamic receiver's class, where the interface of the *resolved* methodblock is located. The JVM first searches the IMT of the receiver's dynamic class for a match to the interface of the *resolved* methodblock. The location of the match is an index, called a guess (for reasons that will be explained later). The IMT entry indexed by the guess contains an array of VMT indexes. The offset in the *resolved* methodblock is used as an offset into this array to obtain the correct VMT index. Figure 3.13 gives the formula for computing the *execution* methodblock from the *resolved* methodblock and the guess. To see why the offset and the index are sufficient, consider a variable input that is declared to be a DataInput and a call-site input.readInt(). The resolved methodblock is readInt() in DataInput. The resolved methodblock has an interface DataInput and a method table offset 1. First, assume that

```
itable = receiver.dynamicClass.IMT[guess];
vmtIndex = itable.vmtIndexArray[resolvedMB.mtoffset];
executeMB = receiver.dynamicClass.VMT[vmtIndex];
```

Figure 3.13: Computing the execution methodblock for invokeinterface.

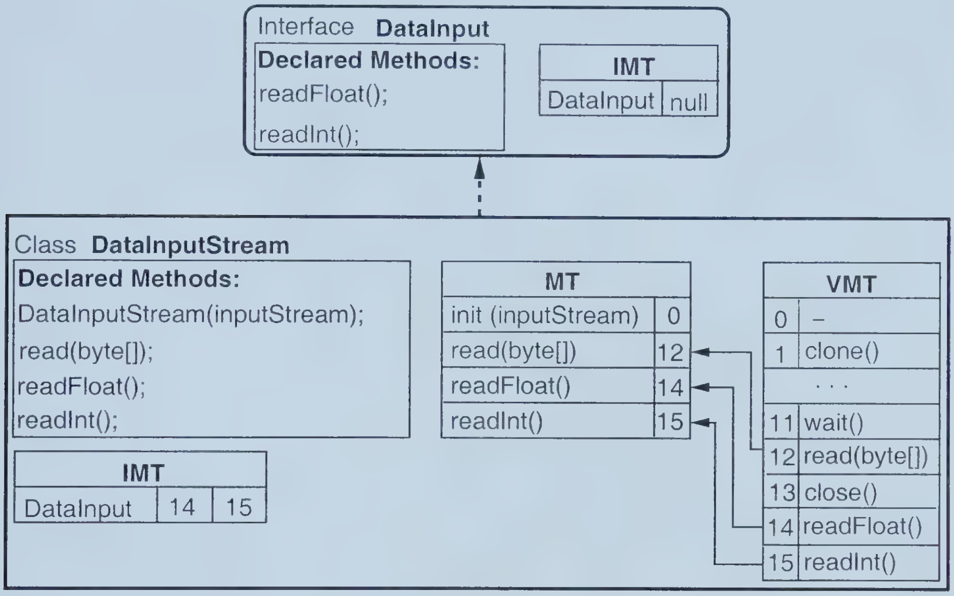


Figure 3.14: Data structures for java.io.DataInputStream.

the receiver object's dynamic class is RandomAccessFile. From Figure 3.12 we can see that a search through the IMT of RandomAccessFile for the interface DataInput produces a guess of 0. The 1st offset of the entry 0 of this IMT is 13 and the VMT entry at index 13 is the right code for readInt(). Alternately, if the receiver object is an instance of DataInputStream, then a search through the IMT of DataInputStream for the interface DataInput also produces a guess of 0, as shown in Figure 3.14. In this case, the 1st offset of the entry 0 of its IMT is 15 and the VMT entry at index 15 is the right code for readInt(). Although the VMT index is not constant across classes (e.g. 13 then 15), the IMT index, together with the array offset, can be used to find the correct code. The IMT provides an extra level of indirection that solves the problem of inconsistent indexing of interface methods between classes. This extra level of indirection is analogous to the way C++ implements multiple-

inheritance using multiple *virtual function tables*. Bytecode quicking modifies the bytecodes at each invokeinterface call-site.

The guess is stored in the quicked bytecodes in addition to a reference to the *resolved* methodblock, so the search does not normally need to be re-done. However, it is possible that the guess at a call-site could be incorrect. To see how the guess can be wrong, consider Figure 3.12, except assume that the IMT in class RandomAccessFile has the DataInput and DataOutput entries in the reverse order. This can happen since classes can implement multiple interfaces, so that the order of interfaces across different IMTs can be different. Re-consider the two successive executions of the call-site input.readInt() described previously. In this case, the first call-site execution (with dynamic class RandomAccessFile) will generate a guess of 1 and an offset of 1. However, when the second execution of this call-site uses the guess of 1, it would be out of bounds in the IMT of DataInputStream. To solve this problem, the interface at the IMT entry with index guess is always compared to the interface of the *resolved* methodblock that is stored in the quicked bytecodes and, if they are different, a new search of the IMT is conducted and the new guess is cached in the quicked bytecodes. This approach is analogous to the inline-caching method-dispatch technique [7] and can suffer from the same thrashing problems if the class of the receiver object of the polymorphic call-site alternates between two classes whose interfaces are stored in different orders. Nevertheless, it is still faster than doing a full resolution from a symbolic method signature for each execution of the call-site.

The details of our modifications are provided later, but in order to support code in interfaces, we change the JVM code that constructs the IMT table in the class loader. To understand our modifications, it is necessary to understand how the class loader currently constructs the IMT table. Figure 3.16 contains the original algorithm for constructing the IMT. We will use the class RandomAccessFile from Figure 3.12 to illustrate the algorithm. The class loader creates a new IMT table (line 1) and then copies the IMT entries of the superclass of the class being loaded to the new IMT table being constructed (line 2). In this example, the superclass of RandomAccessFile is

[illegible]

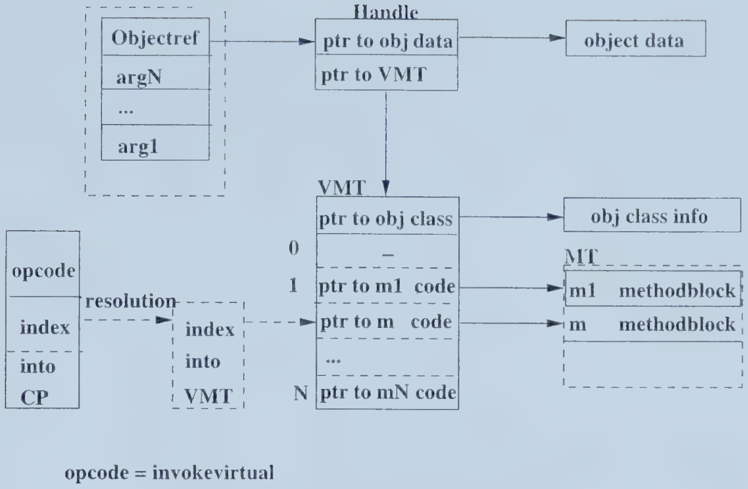


Figure 3.15: Invokevirtual.

`java.lang.Object`, which has no IMT since it does not implement any interfaces, so no entries are copied. The class loader then loops over each interface that is explicitly implemented by the class (line 3). The first interface implemented by `RandomAccessFile` is `DataInput`. The loader fetches the IMT for this interface, which contains entries for `DataInput` and all of the interfaces it inherits (no others in this example). All of the IMT entries that are not already in the new IMT are copied to the new IMT for `RandomAccessFile`, producing a single entry containing (a pointer to) `DataInput`. An associated array that has two slots for indexes (for `readFloat()` and `readInt()`) (line 4) is created. The indexes in this array are not copied since the array does not exist in interfaces. Since `RandomAccessFile` implements a second interface `DataOutput`, the entries in its IMT are also copied down (line 4). Again this is a single entry, but its associated index array has only one entry (for `writeInt(int)`). It is important that each interface is only copied once. For example, if the interfaces `DataInput` and `DataOutput` had a common superinterface `Data`, then, when the IMT entries from `DataInput` were copied, an entry for the inherited `Data` interface would have been included. When the IMT entries for `DataOutput` were copied, the non-unique `Data` interface from `DataOutput`'s IMT would not be copied to the new IMT.

```
Algorithm ConstructIMT(class)
1.  class.imt = new IMT();
2.  copy entries from class.superclass.imt to class.imt;
3.  for each direct interface in class
4.      copy unique entries of interface.imt to class.imt;
5.  end for
6.  for each imtIndex in class.imt
7.      interface = class.imt[imtIndex].interface;
8.      for each mtIndex in interface.MT
9.          imb = interface.MT[mtIndex];
10.         signature = imb.signature();
11.         vmtIndex = class.vmt.findSignature(signature);
12.         class.imt[imtIndex].array[mtIndex] = vmtIndex;
13.         // line reserved for JVM modifications later
14.     end for
15. end for
end algorithm;
```

Figure 3.16: The existing IMT construction algorithm.

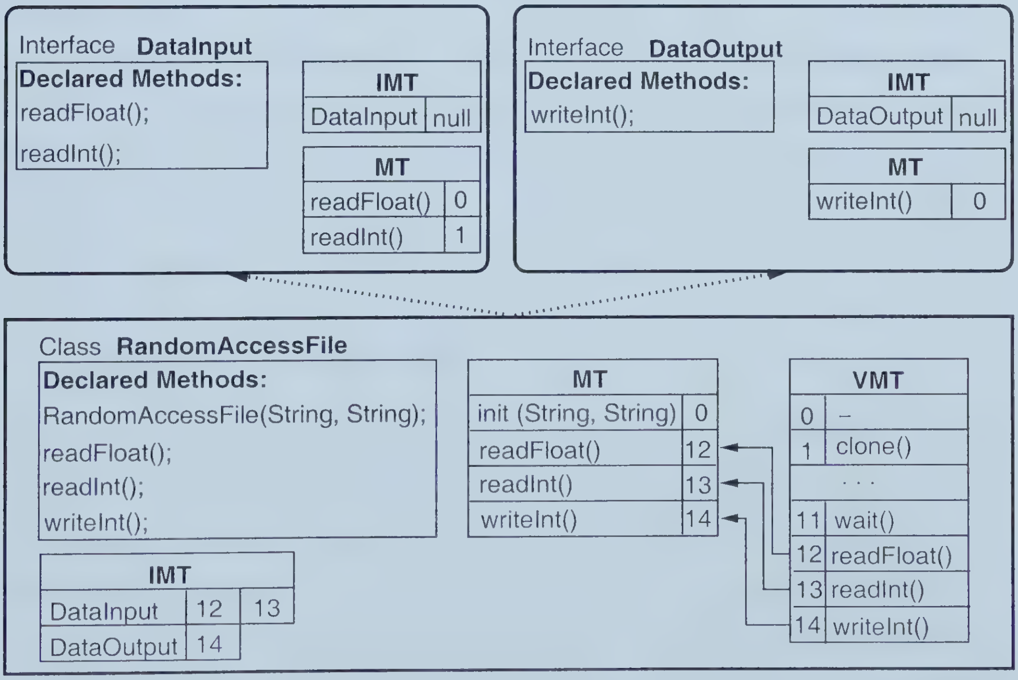


Figure 3.17: Example emphasizing the tables involved in the loading mechanism.

After the class loader is done looping through all implemented interfaces and the IMT has all of its entries, the class loader loops through each slot in the IMT (line 6), to fill in the index arrays. For each slot, the interface pointer is de-referenced to obtain the interface (line 7) and the MT table in that interface is iterated (line 8). For example, the MT table of `DataInput` is iterated first, as shown in Figure 3.17. The entry 0 in the MT is the `methodblock` for `readFloat()` (line 9) and its signature (line 10) is looked up (line 11) in the VMT of the class `RandomAccessFile`. Since a match is found at index 12, this index is copied into the entry 0 of the array in the IMT table for the `DataInput` slot (line 12). The entry 1 in the MT is `readInt()` and when it is looked up in the VMT (line 11), the index found is 13. The index 13 is copied into the IMT array at the `DataInput` slot's array index 1 (line 12). The process continues until all arrays at all IMT slots are full.

The Sun JVM makes a simple optimization to the code shown in Figure 3.16. The loop in step 6 of the optimized code does not iterate over all IMT indexes. Instead, it starts at the first index after the entries that were copied from the superclass's IMT. This is possible since the array indexes in the entries of the superclass will not change in the class being loaded, so these entries are simply copied instead of being calculated. However, we iterate over all indexes of the IMT to support the changes described in Chapter 4. Since this code is run only at class load time, the performance loss is insignificant.

3.7 Quick Bytecodes

An analysis of the dispatch process for both bytecodes, `invokevirtual` and `invokeinterface`, shows that the interpretation of the bytecodes used to invoke methods can be improved. Opcodes that refer to CP entries can be replaced by `_quick` opcodes after the CP references are resolved. Replacing the normal opcodes with `_quick` counterparts in the bytecode stream can substantially speed up their interpretation. When the JVM encounters a `_quick` instruction, it knows that the entry has already been resolved, so it can execute the instruction faster. In some cases, the operands are overwritten with

data representing a direct reference. The details of quicking `invokevirtual` and `invokeinterface` are different.

Invokevirtual has a single operand, which is a two-byte integer index into the run-time constant pool, where the method signature is stored. `Invokevirtual` has three quick opcodes: `invokevirtual_quick`, `invokevirtualobject_quick`, and `invokevirtual_quick_w`.

1. For `invokevirtual_quick`, the original two-byte operand is replaced by a one-byte `offset` into the VMT and one byte that stores the number of arguments, `nargs`, as illustrated in Figure 3.18. This second byte is needed to find the receiver object on the stack. The number of arguments was previously computed after obtaining the method signature from the constant pool. The JVM uses the number of arguments to reach the receiver and follow its pointer to the VMT. To use `invokevirtual_quick`, the index into VMT must be 255 or less and the dynamic class of the receiver object cannot be an instance of class `java.lang.Object`.

2. `Invokevirtualobject_quick` has the same operands as the previous bytecode, `invokevirtual_quick`. It is used for invoking instance methods of class `java.lang.Object` and it is introduced specifically for arrays. The *objectref* on the operand stack is a reference to an object or to an array. The `offset` retrieved from the operand stack is an index into the VMT of `java.lang.Object` and ultimately indicates the right `methodblock`.

3. `Invokevirtual_quick_w` is followed by the same two-byte index into the constant pool as the unquicked `invokevirtual` bytecode. With the `_w` variation, the constant pool entry is changed, instead of the bytecode operands. This `_quick` opcode is used when the index in VMT is greater than 255. Method resolution simply replaces the method signature in the run-time constant pool with an entry containing a two-byte index into the VMT and one byte that represents the number of method arguments, `nargs`.

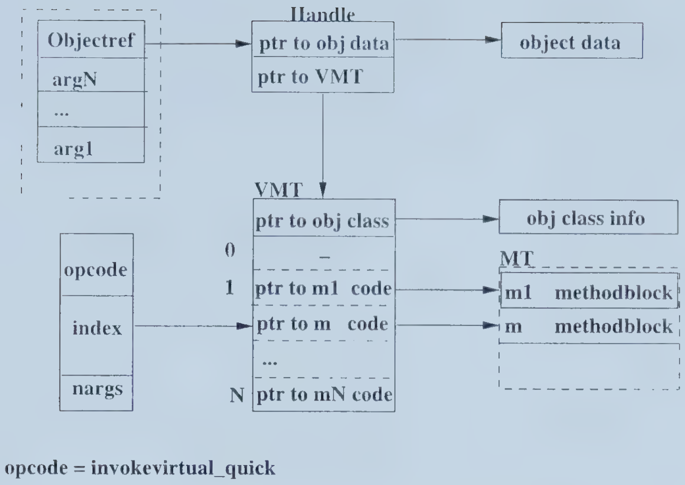


Figure 3.18: Invokevirtual_quick.

Invokeinterface has only one `_quick` counterpart used for the invocation of interface methods: `invokeinterface_quick`. This bytecode is similar to `invokevirtual_quick_w` in that the original operand index into the run-time constant pool is retained. However, the run-time constant pool entry that it points to is changed to point to the `methodblock` that was resolved when the call-site was first executed. In addition, two other operands are added to the bytecodes, a `guess` and the number of method arguments, `nargs`. The `guess` is an index into the IMT that specifies one of the implemented interfaces. There is an array at that IMT slot for indexes into the VMT of all methods declared in an interface, as described earlier in this Chapter. To obtain the appropriate index into that array, the first operand is used to obtain the `methodblock` from the run-time constant pool and the `methodblock` contains the required index into the array. However, the `guess` operand is called a guess for an important reason. It is possible that it indexes the wrong interface in the IMT. Before the index is retrieved from the `methodblock`, the interface of the `methodblock` (stored as a pointer field in the `methodblock`) is compared to the interface pointer contained at the guess index of the IMT. If they are the same, the index from the `methodblock` is used. If they are different, then the `guess` is wrong and the correct interface must be found. In this case, the IMT

is searched for the interface pointer that matches the interface pointer stored in the `methodblock`. Once one is found, the guess operand is changed to the new index in the IMT and the dispatch continues.

3.8 Concluding Remarks

In this Chapter, we described the current implementation of the original JVM's data structures involved in method dispatch.

The two steps of method dispatch, resolution and execution, were detailed for the `invokevirtual` and `invokeinterface` bytecodes. Since resolution is quite slow, we described bytecode quicking which modifies the bytecodes at each resolved call-site to run faster on subsequent execution times of the call-site.

In the next Chapters, we will discuss how the method dispatch in the JVM is modified.

Data structure name	Data structure title	Description Summary
`methodblock`	Method block (mb or imb)	Stores the complete information (including code) about a method. It includes the size of the operand stack and local variable sections of the method's stack, a pointer to the method's bytecodes, the method signature, and an exception table.
`methods`	Method Table (MT)	Array of `methodblocks`. Both classes and interfaces have MTs. It has an entry for every method declared (**not** inherited) in the class or interface. Therefore, it contains `methodblocks` for all overriding methods.
`methodtable`	Virtual Method Table (VMT)	Array of pointers to `methodblocks`. Only classes have VMTs. Each slot (entry) holds a reference to an instance method implementation that has been *declared* or *inherited* by the current class.
`imethodtable`	Interface Method Table (IMT)	Array of structures which contain information about interfaces. Both classes and interfaces have IMTs. Every *class* has an IMT that references all of the interfaces it implements or inherits; every *interface* also has an IMT with slots for all the interfaces it extends, including itself. The IMT provides an extra level of indirection that solves the problem of inconsistent indexing of interface methods among classes.

Table 3.1: Major data structures involved in method dispatch.

Chapter 4

Implementation

This Chapter presents the details of the JVM modifications to accommodate code within interfaces and describes the simple test cases that are used to verify the implementation of multiple code inheritance. Neither the syntax of the Java programming language nor the `javac` compiler are changed. A scripting process was developed instead of using syntax changes. The details of the scripting process are presented in Chapter 7.

4.1 Our Approach

Our implementation of multiple code inheritance in Java is based on the novel concept of adding code to selected interfaces. We show that only straightforward and localized modifications are made to the JVM to support code within the interfaces.

If code is put into interfaces and an existing Java compiler is used, compilation errors can occur. For example, if the code for `readInt()` in Figure 3.14 is moved from the class `DataInputStream` to the interface `DataInput`, an unmodified compiler would not compile the code in `DataInput` and would complain that there is no method declaration for `readInt()` in the `DataInputStream` class, therefore the class must be declared as abstract. Since it requires considerable engineering effort, we have not modified a Java compiler to recognize code in interfaces. Instead, we have created a scripting process that allows a programmer to insert method code into interfaces and work around a standard compiler. Details about the process and the tools that support our approach

```
13A.if (imb.code <> null) // code in interface
13B.    currentmb = class.vmt[vmtIndex];
13C.    if (currentmb.code == null) // no code in MT
13D.        class.vmt[vmtIndex] = imb; // point VMT to imb
13E.    else // potential code ambiguity
13F.        if ((!  currentmb.class.imt.contains(imb)) &&
13G.            (!imb.class.imt.contains(currentmb)))
13H.            throw ambiguous method exception
13I.        end if
13J.    end if
13K.end if
```

Figure 4.1: Code added to Figure 3.16 to support interface code.

can be found in Chapter 7. In this Chapter we assume that this process is used to put code into interfaces and the compiler does not generate any compilation errors due to missing method declarations. We have taken this approach because we want to quickly test the utility of code in interfaces to support multiple code inheritance, without the full-fledged engineering effort of modifying a compiler.

4.2 JVM Modifications

To support code in interfaces, we modified the JVM code that constructs the IMT table in the class loader [21] (Figure 3.16), as shown in Figure 4.1. After a VMT index is inserted into the array of an entry in the IMT table, the corresponding VMT table entry is checked. If the VMT table points to a `methodblock` in an MT that has no code, then the VMT table entry is changed to point to a `methodblock` in the MT of the interface that contains the code as shown in Figure 4.1. However, it is possible that the method code is ambiguous, as we will discuss further in this Chapter, Section 4.4.

We use the class `RandomAccessFile` as an illustrative example. Assume we are loading this class. Also assume that the code for `readInt()` is moved to the `DataInput` interface instead of being in the `RandomAccessFile` class, as shown in Figure 4.2. Assume the array element at index 1 of the `DataInput` entry is set to 13 (line 12 in Figure 3.16). Therefore, the `imb` is bound to

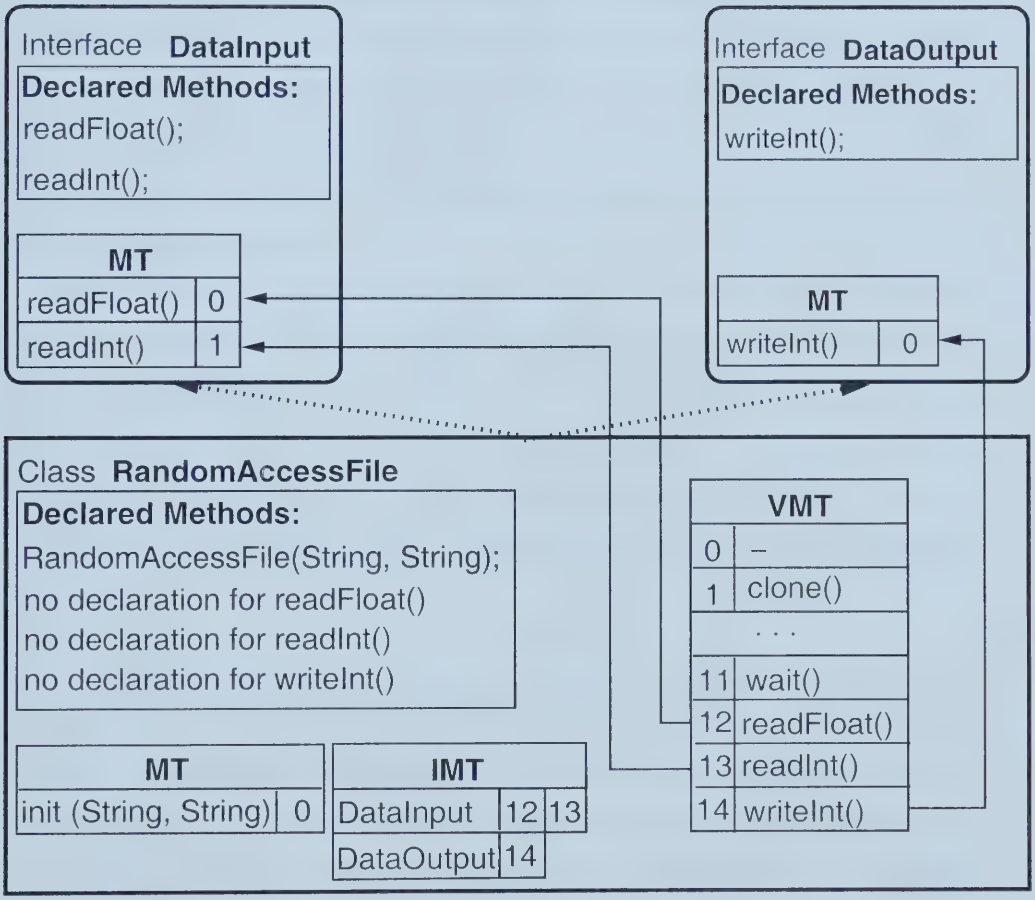


Figure 4.2: The code from `java.io.RandomAccessFile` is moved up in two of its direct superinterfaces.

the readInt() methodblock in DataInput and this methodblock has code (step 13A in Figure 4.1). Our modified class loader accesses the VMT entry of RandomAccessFile at index 13 to obtain the current methodblock for readInt() from the MT of RandomAccessFile (step 13B). Since there is no code in the current methodblock, the code pointer is null (step 13C). Therefore, we change the VMT entry at index 13 to point to the methodblock in DataInput instead. Note that the IMT offset into VMT stays the same, it is only the slot in VMT that is modified. The resolution and dispatch of invokevirtual proceeds in exactly the same way as with the unmodified JVM, but the change in the class loader code allows the code in the interface to be found and executed.

We also needed to modify the dispatch in the situation where a call-site such as this.alpha() appears inside an interface method. In this case, the call-site is turned from an invokevirtual to an invokeinterface, because the static type of this is an interface. With the design choices we made, no other changes were required to support code in interfaces (and hence multiple-inheritance) and this is due to the Miranda Methods concept incorporated in SUN JVM.

4.3 Exploiting Miranda Methods

If there is a declaration for readInt() in DataInput, this method must be understood by any of the classes which implement DataInput. Therefore, the VMT of each of these classes must have a slot for readInt(). The slots can be obtained by either of the following two methods.

The javac compiler generates methods in each abstract class for all interface methods that are contained in all interfaces that are implemented by the class, but which do not have an implementation in the class. Such generated methods are called *Miranda methods*, because if the class does not have a corresponding interface method, one is provided by default. For example, the entry in the VMT of RandomAccessFile for method readInt() of Figure 4.2 is a Miranda method since there is no code (no explicit declaration) for

`readInt()` in `RandomAccessFile`. These methods are added because early VMs did not look for methods along the interface path, performing the lookup only along the superclass chain.

However, if a compiler does not generate Miranda methods, one additional action is required at class load time in our implementation. For each interface from a class's IMT, loop through their methods and look for corresponding methods in that class's VMT. If one is not found. then extend that class's VMT with this method and make it point to the code in the current interface method. In both cases, the newly created slot in the VMT is present in all the sub-classes of that class, therefore if code is found in one superinterface, it will be propagated to all the classes implementing that interface.

4.4 Inheritance Scenarios - Potential Ambiguities

We have analyzed situations that use code in interfaces to ensure that the algorithm in Figure 4.1 works as necessary. The four scenarios in Figure 4.3 represent the common situations. They test all paths of the algorithm we devised. The first scenario shows a non-ambiguous case. The second scenario illustrates a simple method overriding case with no ambiguities. The third scenario generates an ambiguity, since a type inherits implementations for a method from two direct unrelated parents; note that the type itself does not provide an implementation for that method. Finally, the fourth scenario is a case of complex method overriding which does not generate an ambiguity under a weaker definition of inheritance conflicts.

Scenario 1. The simplest scenario occurs when `ClassA` has no code for method `alpha()` and no superclass has code for method `alpha()`. In addition, a direct superinterface, `InterfaceA`, has code for method `alpha()`. This is also the scenario described previously, where the class `RandomAccessFile` implements the interface `DataInput` which contained code for `readInt()`. When a message `alpha()` is sent to an instance of `ClassA`, the code from `InterfaceA` is executed.

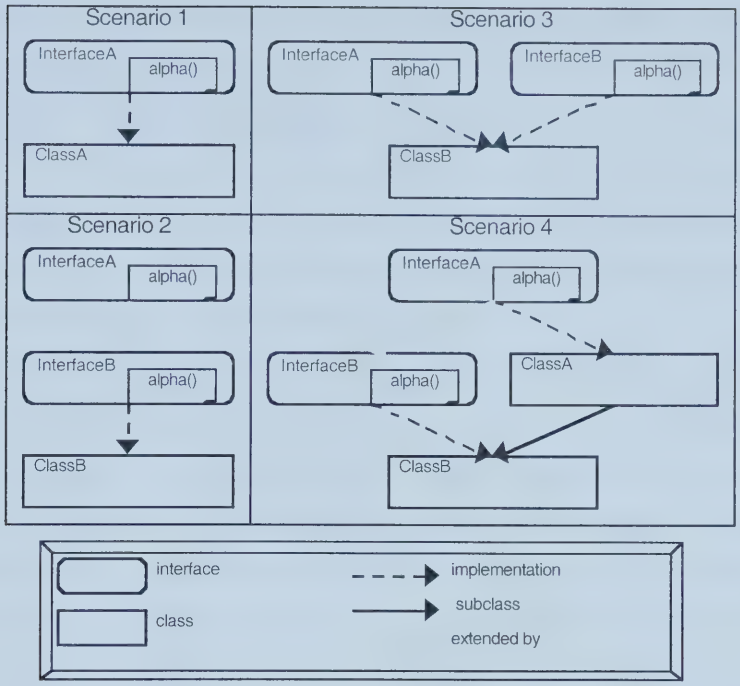


Figure 4.3: Inheritance scenarios - Potential ambiguities.

Scenario 2. A more complex case occurs when `ClassB` has no code for `alpha()`, but both `InterfaceA` and `InterfaceB` on the same superinterface chain have code for `alpha()`. In this case, step 13C of Figure 4.1 is first executed with `currentmb` bound to a `methodblock` in `ClassB` (with no code) and `imb` bound to a `methodblock` in `InterfaceB` with code. This means that step 13D is executed to re-bind the VMT entry to the `methodblock` in `InterfaceB`. The second time that step 13C is executed, `currentmb` is bound to a `methodblock` in `InterfaceB` (with code) and `imb` is bound to a `methodblock` in `InterfaceA` (with code). Step 13F is entered since there is chance for method ambiguity. However, since `InterfaceA` is a superinterface of `InterfaceB`, the condition in step 13F evaluates to false and an ambiguous method exception is not thrown. Therefore, when a message `alpha()` is sent to an instance of `ClassB`, the code for `alpha()` provided by `InterfaceB` is executed. This constitutes a simple method overriding situation, similar to the case where we have classes instead of interfaces.

Scenario 3. This scenario illustrates a situation where an ambiguous method exception should be thrown. Since either the code in InterfaceA and InterfaceB could be inherited, the programmer is required to declare a method in ClassB to resolve the inheritance conflict [14]. A trace of the code in Figure 4.1 shows that an ambiguous method exception does occur since the IMT for InterfaceA does not contain InterfaceB and the IMT for InterfaceB does not contain InterfaceA.

Scenario 4. This scenario illustrates an interesting situation where one might conclude that an ambiguous method exception should be thrown for an alpha() in ClassB. However, since the code for alpha() in InterfaceA is reachable from ClassB by going through InterfaceB, a weaker definition of inheritance conflict would dispatch the version of alpha() from InterfaceB [24]. A trace of the code in Figure 4.1 shows that an ambiguous method exception does not occur since the condition in step 13G is false. In this case, currentmb is bound to a methodblock in InterfaceA and imb is bound to a methodblock in InterfaceB, since interfaces of superclasses are added to the IMT of ClassB before other interfaces are added, as described in the code in Figure 3.16. Therefore, in this situation, the code from InterfaceB is executed when a message alpha() is sent to an instance of ClassB. Each scenario illustrates one of the unique paths through the code in Figure 4.1, including the need for both conditions (step 13F and 13G).

4.5 Dispatch of Code from Interface Methods

When the user provides interfaces with code, a call-site that often appears in an interface method is this.alpha(). In a method implemented in a class, the this keyword represents a reference to the object on which the method was invoked and an invokevirtual bytecode is generated. When this call-site is found in an interface, an invokeinterface should be generated instead since the static type of this is now an interface. To account for such situations, we modify the bytecode generated for each this.alpha() call-site found in an interface from invokevirtual to an invokeinterface. Thus, the lookup

```
// java.io.DataInputStream and java.io.RandomAccessFile
...
public final byte readByte() throws IOException {
    int ch = this.in.read(); // int ch = this.read();
    if (ch < 0)
        throw new EOFException();
    return (byte)(ch);
}
...
```

Figure 4.4: Similar code in java.io library.

for the method starts in the current interface (the interface that contains the call-site) and continues up its superinterface chain searching the method table of each interface for a method signature match.

We present an example of such a method dispatch that we have encountered in the validation process of our JVM modifications. Figure 4.4 is a reproduction of Figure 2.9 that illustrated similar code in the readByte() methods of the `java.io` library from the classes `DataInputStream` and `RandomAccessFile`.

If this code can be made identical, it can be moved to the common superinterface of `DataInputStream` and `RandomAccessFile`, called `DataInput` that is shown in Figure 2.7. The code can be made identical by replacing the second line of the `readByte()` method by:

`int ch = this.source();`

where the `source()` method in class `DataInputStream` returns `this.in` and the `source()` method in class `RandomAccessFile` returns `this`. This change is described in more detail in Chapter 6. What is important to notice now is that this kind of abstraction results in a method such as `readByte()` in the interface `DataInput` which contains a message with `this` as the receiver. In this case, the scripting process replaces the `invokevirtual` bytecode with an `invokeinterface` bytecode.

The operands required by `invokevirtual` and `invokeinterface` are different. The `invokevirtual` bytecode takes only two operands which form an index into the constant pool, whereas `invokeinterface` takes four operands: the first two operands form an index into the constant pool, the third operand

indicates the number of arguments that the method takes and the fourth is set aside for execution speed (the `guess`) after quicking. Therefore, when replacing these bytecodes in the `.class` file, the number of arguments should be added (if the last operand is not provided, it is automatically set to 0). A new `.class` file containing these changes is generated as described in Chapter 7. The number of arguments provided in the script is not important, because as we will see later we do not use it. Instead, the correct number of arguments for the method is taken from the resolved `methodblock` when the JVM encounters an `invokeinterface`. The right number of arguments is essential, because when the method that we would like to execute is in an interface, the receiver object has to be retrieved by going up the operand stack a number of arguments found in the resolution `methodblock`, and not in the bytecodes (which do not reflect the actual situation in the resolved `methodblock`).

4.6 Concluding Remarks

Our implementation of multiple code inheritance in Java is based on the novel concept of adding code to selected interfaces represented by code-types. In this Chapter, we described our modifications to the JVM class loader that support code within interfaces and we showed how the new code was dispatched. We showed that our approach detects ambiguous situations due to code in multiple super-types. We illustrated scenarios that tested the modified class loader in the presence of code in interfaces, as well as in ambiguous situations.

We solved the special dispatch problem for `this.alpha()` call-sites within interface code, by replacing the `invokespecial` bytecode, normally generated, with `invokeinterface`.

We provided a comment notation for including code in interfaces. In Chapter 7 we detail the process which inserts code within interfaces in the absence of compiler support for multiple code inheritance.

Chapter 5

Super Call Implementation

The implementation of an overridden method often invokes the same method as implemented in the super-type, in order to refine existing code. Java achieves this enhancement of functionality for the overridden method through its powerful super call mechanism. However, when a type inherits code from multiple super-types, the choice of which type to use for a super call becomes an issue. In this Chapter we present a solution to the problem of super calls in the case of multiple code inheritance. We also propose syntax changes for super calls to interfaces that would simplify coding.

5.1 Super Call Mechanism

In Java, a method invokes the same method from its superclass using the syntax `super.alpha()`. With multiple-inheritance, such a call could be ambiguous. C++ solves this ambiguity problem by specifying a method in a particular superclass at compile time. For example, if `C` is a direct subclass of classes `A` and `B` that both declare method `alpha()`, then a super reference to `alpha()` in a method in class `C` can specify either `A::alpha()` or `B::alpha()`. In fact, if no declaration of `alpha()` occurs in class `A`, but does occur in a superclass of `A`, such as `D`, then the call `A::alpha()` would start a dynamic lookup in `A` and then proceed to find the appropriate method in `D`.

This is the approach we use in multiple-inheritance Java. Chapter 7 describes the syntax used to implement this idea without changing the Java language. In this Chapter we use the simple notation `super(Start).alpha()`,

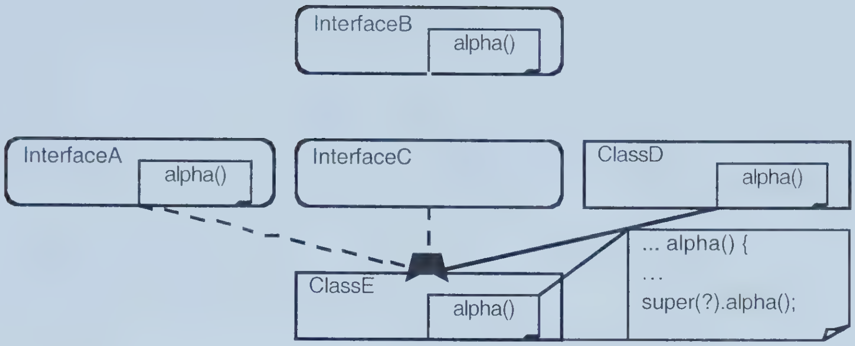


Figure 5.1: Classes and interfaces for super calls.

where `Start` refers to any superinterface. Since the argument interface does not need to declare the method, this argument indicates the place where the lookup starts. The modified JVM looks for code in the specified interface and then continues searching along the superinterface chain. If a stricter form of multi-super is required, the start interface could be restricted to be an immediate superinterface of the class or interface that includes the super call. Some would argue that this C++ model provides too much freedom in super calls.

5.2 Examples of Multi-Inheritance Super

Consider the classes and interfaces in Figure 5.1. The following method call `super(InterfaceA).alpha()` in a method of `ClassE` invokes the `alpha()` in `InterfaceA`. The call `super(InterfaceC).alpha()` invokes the `alpha()` in `InterfaceB`. The call `super.alpha()` invokes the `alpha()` in `ClassD`, because we do not change the meaning of the single-inheritance super call.

Now consider the interfaces and classes of Figure 5.2. The method call `super(InterfaceG).alpha()` in a method of `ClassM` invokes the `alpha()` in `InterfaceF`. The call `super.alpha()` would behave identically with the usual super call. The traditional call `super.alpha()` would not find the `alpha()` in `InterfaceJ` and in fact would result in a compile-time error since there is no `alpha()` declared in the superclass chain of `ClassM` (i.e., there is no declaration of `alpha()` in `ClassL` nor `ClassK`). If there was also an `alpha()` in `ClassK`,

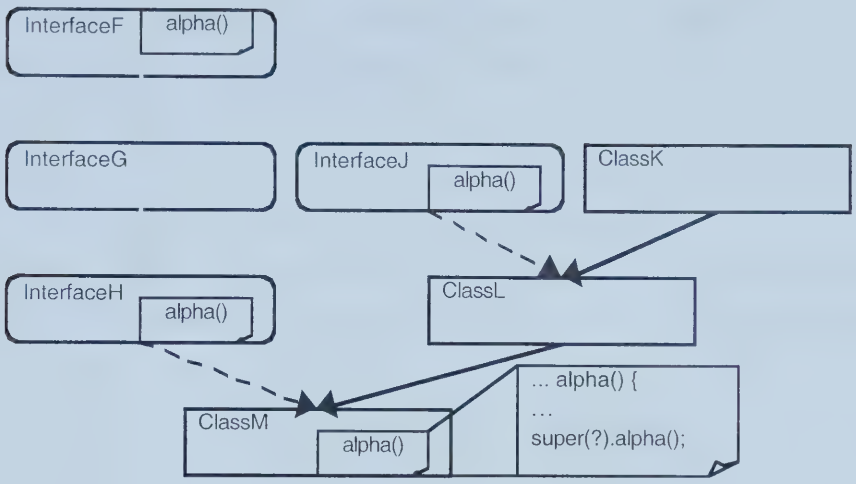


Figure 5.2: More classes and interfaces for super calls.

then an inheritance conflict exception would have been thrown when `ClassL` was loaded.

5.3 Implementation of Super

Our implementation of the multiple-inheritance super call, with the proposed syntax `super(Start).alpha()`, generates an `invokeinterface` bytecode instead of an `invokespecial` bytecode generated to implement a single-inheritance super call, `super.alpha()`.

The instruction stores the argument *interface* (in this case, `Start`) in the constant pool and marks the method call-site by storing a special value in an operand of `invokeinterface` bytecode. With compiler support, we would prefer to create a different bytecode (`invokemulti-super`).

We will refer to this operation as an `invokemulti-super`, even when it is represented by a marked `invokeinterface`. It appears that two JVM changes are required to support `invokemulti-super`, one in *resolution* and one in computing the *execution* `methodblock`. In fact, *resolution* does not require changes. Regular `invokeinterface` resolution finds an appropriate *resolution* `methodblock`. However, *execution* `methodblock` computation is different

for `invokemulti-super` and `invokeinterface`. An `invokeinterface` uses a `guess` and the *resolved* `methodblock` stored in the quicked bytecodes to find the *execution* `methodblock` using the formula given in Figure 3.13. For `invokemulti-super`, we can just use the resolved `methodblock` directly as the execution `methodblock`, as we show later in this Chapter.

We preserve the semantics of the traditional super calls without altering their performance. If we use the normal super syntax (without any argument for super), the classic super would be executed, therefore the code from the superclass would be retrieved. The compiler emits an `invokespecial` bytecode followed by one operand, which is an index into the constant pool of the current class. The entry at this index is the method signature (in this case, `alpha()`) of the method being invoked, along with the first superclass that contains a declaration of `alpha()`, when the super call was compiled.

If an existing program contains a super call, we expect the new JVM to generate the same results. This is consistent with Figure 5.2 where a regular `super.alpha()` call in `ClassM` generates a compiler error instead of executing the code in `InterfaceJ`. If on the other hand we use the multi-super syntax with an interface argument, then we expect the code from an interface to be executed.

If the user wants the code from a specified superinterface to be executed, then the name of the superinterface has to be supplied as an argument to the multi-super. In this case, the scripting process applied to the interfaces and classes involved recognizes the special marker (from the number-of-arguments operand of `invokeinterface`) and replaces the static receiver of the method with the specified interface name.

We have slightly modified the JVM code that executes the `invokeinterface` bytecode in order to differentiate between the two cases when this bytecode is generated: the traditional case (real `invokeinterfaces`) and this special case (the multi-inheritance super call). In fact, we have modified the `_quick` counterpart of the `invokeinterface`, i.e., `invokeinterface_quick`. In the case the flag set in the operand is on, when an `invokeinterface_quick` bytecode is executed, our execution `methodblock` is retrieved, instead of the usual

```
case opc_invokeinterface_quick:
    imb = constant_pool[GET_INDEX(pc + 1)].mb;
    interface = imb.class;
    offset = imb.offset;
    ...
    // We change the code so that nargs is retrieved from
    // the resolved interface methodblock imb instead of
    // from the nargs bytecode (pc[3]).
    // args_size = pc[3]; // REMOVED
    args_size = imb.args_size; // ADDED
    optop -= args size;
    ...
    // We use the third operand (nargs) as a marker for the
    // multi-super case.
    if (pc[3] == 255) // ADDED
        mb = interface.MT[offset]; // ADDED
        goto callmethod; // ADDED
    end if // ADDED
    ...
end case
```

Figure 5.3: The modifications made at the invokeinterface_quick bytecode execution.

methodblock. Details of our implementation are illustrated in Figure 5.3. This does not affect the non-marked invokeinterface executions, because if the flag is not set, then the next time an invokeinterface is encountered, the usual (non-modified) execution process occurs and the proper methodblock is executed. The execution of the traditional invokeinterface_quick is more complicated, since the resolution methodblock may be different from the execution methodblock as described in Chapter 3, Section 3.7.

In our case, it turns out that the resolution methodblock is the actual execution methodblock. The reason for this convenient situation is that once we specify the starting point of the lookup (the interface argument) the interface method table (IMT) of the specified interface is searched, beginning with its first entry, which is the interface itself, and continuing with all the direct and indirect superinterfaces until a matching signature for the method is found in the method table (MT) of some interface. The resolved method is the method

that should be executed since in a super call the dynamic type of the receiver is irrelevant.

If no code is found on the superinterface chain, the compiler would have generated an error. Note that it is currently possible to use our scripts to put code in a method `alpha()` in an interface `IA` and to declare `alpha()` to be abstract in a sub-type interface `IB`. This should be a compile-time error since it violates substitutability [14]. Because we currently do not have compiler support for code in interfaces, we catch this error at load-time.

5.4 Concluding Remarks

In this Chapter, we presented the JVM changes necessary to support our generalization of the super operation for multiple inheritance. We defined and implemented a super call mechanism that resembles the one in C++. We achieved this by making a change to the execution of the `invokeinterface` bytecode.

We provided a simple notation for super calls to interfaces, which does not require compiler support. In Chapter 7, we detail the scripting process used to work around the standard Java compilers in the presence of multiple code inheritance. We proposed syntax changes for super calls to interfaces that would simplify coding.

Chapter 6

Experimental Results

This Chapter provides an overview of experiments and tests conducted during the process of verifying the implementation of our SUN JVM for JDK 1.2.2 modifications. The goal of our JVM validation is to show that our multiple code inheritance implementation preserves semantics and performance of existing single inheritance code, without altering Java language syntax or Java compilers. In addition, we show that both our basic multiple code inheritance and the super call mechanism we implemented execute correctly in multiple inheritance programs. We also provide some measurements of the software engineering advantages of using multiple code inheritance.

6.1 Experimental Platform

The experiments were executed on an Intel PC, single Pentium III processor 700MHz, with 256 KB L2 cache size and 512 MB RAM. We compiled the Sun Microsystems JDK 1.2.2 for the Linux v. 2.2.16-3 operating system with the GCC compiler v. egcs-2.91.66 with optimization flags -O2 (default) in JVM internal debug mode based on conditional compilation. This JVM version does not have a `jit` compiler. We developed a scripting process using Perl v5.6.0 for Linux.

6.2 Compatibility and Performance

We ran two large single-inheritance Java programs on the unmodified JVM and on our modified JVM. We wanted to test that our modified JVM did not introduce errors into single-inheritance programs.

The single inheritance test programs were `javac` and `jasper`. The `jasper` application takes a `.class` file and turns it into a `.j` file containing a human readable version for the binary code of a `.class` file. In the first experiment we compiled all of the files in the `java.io` package. In the second experiment we applied `jasper` to all of the `.class` files in the `java.io` package. Both `javac` and `jasper` are written in Java, so they require a JVM to run.

In order to check if the results were consistent, we compared with the Unix command **`diff`** the binary files produced by the `javac` compiler ran on the classic JVM against the `javac` compiler ran on our modified JVM, and we verified they are identical. We also verified that the outputs of `jasper` are identical when ran on the two JVMs.

We repeated this experiment for `javap` (Chapter 3, Section 3.1 illustrates an example of using this tool), a single inheritance application within the JDK, which generates a description of any `.class` file that is provided as an argument. We tested the `.class` file disassembler `javap` on the `.class` files generated by `javac` in the `java.io` library. Again, the output using our modified JVM is identical to the output using the classic JVM.

We also wanted to measure the performance overhead of using our modified JVM on single inheritance programs.

In all three of these tests, there is no measurable change in the execution times, the performance is the same within measurement errors. Table 6.1 shows the average times (seconds) obtained with the Unix command `time`, after 20 runs of `javac` and `jasper`. The table shows also the corresponding standard deviation with both JVM implementations. No times are included for `javap` since it only runs on a single `.class` file and the time is too short for meaningful comparisons.

[illegible]

JVM	javac	jasper
Sun JVM avg	10.25s	11.85s
Our JVM avg	10.08s	11.56s
Sun JVM stdev	0.02	0.54
Our JVM stdev	0.15	0.05

Table 6.1: Time measurements for `javac` and `jasper` on `java.io` library files.

6.3 Correctness

We then ran programs whose inheritance structures are represented in Figure 4.3, to test the basic implementation of multiple-inheritance. This includes code in interfaces and inheritance of this code. These situations test all paths through our modified class loader code shown in Figure 4.1.

We also included tests for the special call-sites `this.alpha()` in an interface method code. Also, we included tests for `input.alpha()` call-sites within an interface with code, where `input` is declared to be that interface. These call sites would normally be compiled into `invokevirtual` bytecodes as a result of applying our scripting process. We turn them into `invokeinterface` bytecodes. In all cases, we obtained the expected results described in more detail in Chapter 4.

To test our implementation of super calls, we ran programs with all of the inheritance structures of Figure 5.1 and Figure 5.2. We tested the execution of the traditional super calls when code is provided in superinterfaces and the execution of multiple inheritance super calls. The results demonstrate that the semantics of traditional super calls are preserved and that multiple inheritance super calls are correctly dispatched, as compared to the expected results discussed in Chapter 5.

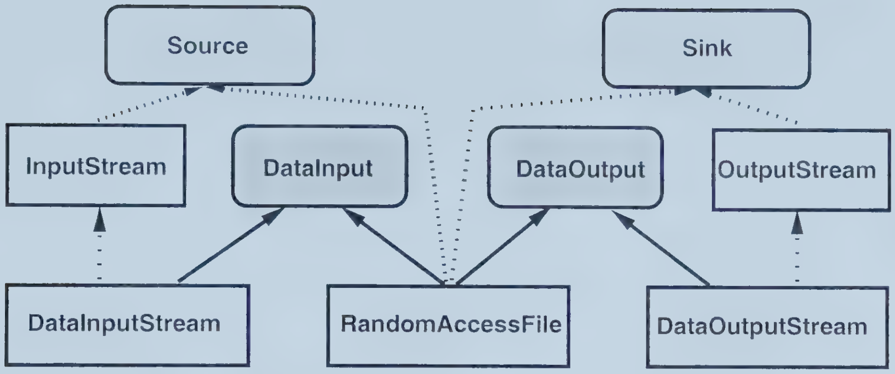


Figure 6.1: Re-factored hierarchy in java.io library.

```
// java.io.DataInputStream and java.io.RandomAccessFile
public final float readFloat() throws IOException {
    return Float.intBitsToFloat(readInt());
}
```

Figure 6.2: Identical code in the input stream files.

6.4 Re-factoring the java.io Library

One of the common examples which motivates the use of multiple code inheritance is the java.io library. Identical code appears in several classes within this library. Figure 6.1 shows the existing hierarchy of classes and interfaces, along with two new interfaces, Source and Sink, that are used to help promote code to superinterfaces.

6.4.1 Input Stream Classes

Classes RandomAccessFile and DataInputStream have either identical code or code that requires a simple abstraction in order to be made identical. The goal is to promote the common code into the DataInput interface where it would be available for instances of both classes. For example, the method readFloat() from Figure 6.2 has the same code in both classes. The method readByte() from Figure 6.3 needs one abstraction. We accomplish that by replacing references to data by abstract accessor method invocations (e.g. source() and sink() as discussed below) placed in the interfaces and imple-

```
// java.io.DataInputStream and java.io.RandomAccessFile
public final byte readByte() throws IOException {
    int ch = this.in.read(); // int ch = this.read();
    if (ch < 0)
        throw new EOFException();
    return (byte)(ch);
}
```

Figure 6.3: Similar code in the input stream files.

```
// java.io.DataInput
public final byte readByte() throws IOException {
    int ch = this.source().read();
    if (ch < 0)
        throw new EOFException();
    return (byte)(ch);
}
```

Figure 6.4: Abstraction of similar code in `DataInput` interface.

mented in the classes down the hierarchy. Figure 6.4 shows how we abstract the code and promote it to the common superinterface `DataInput`.

Let us consider the code in the `readByte()` method from `DataInputSteam` and `RandomAccessFile` shown in Figure 6.3. Both methods call the method `read()`. The only difference between the code in the classes `DataInputStream` and `RandomAccessFile` (which implement `DataInput`) is the receiver of the `read()` method. To generalize the code for this method so that it can be promoted to the interface `DataInput`, we have to declare a method `source()` in the interface `DataInput` which returns the right receiver for the `read()` method in each case. The implementations of the `source()` method for classes `DataInputStream` and `RandomAccessFile` are shown in Figure 6.5. Since the `source()` method in `DataInputStream` returns an instance of class `InputStream` and the `source()` method in `RandomAccessFile` returns an instance of class `RandomAccessFile`, we need a smallest common super-type of `InputStream` and `RandomAccessFile`. Therefore we introduce a new interface `Source`, as shown in Figure 6.1. In the same manner, we need a `sink()` method declared in `DataOutput` and a `Sink` interface, as we will see in the

```
// class java.io.DataInputStream
...
    public Source source() {
        return this.in;
    }
...
// class java.io.RandomAccessFile
...
    public Source source() {
        return this;
    }
...
```

Figure 6.5: Implementation of the `source()` method.

```
// package mi
    public interface Source {
        public int read() throws IOException;
    }
    public interface Sink {
        public void write(int b) throws IOException;
    }
```

Figure 6.6: The `mi` package.

next Section.

We have re-factored the `java.io` library by moving common code up into interfaces. To support this process, we have built a package named `mi` (Figure 6.6) that is imported in every class or interface that implements or extends our two new interfaces: `Source` and `Sink`. The `Source` interface has one abstract method, `read()` and `Sink` interface has one abstract method, `write(int)` as illustrated in Figure 6.6. These two interfaces represent the least common superinterface of the types returned by the `source()` and `sink()` methods.

`InputStream` and `RandomAccessFile` both implement the new interface `Source`, and at the same time `OutputStream` and `RandomAccessFile` both implement the new interface `Sink`.

```
// java.io.DataOutputStream and java.io.RandomAccessFile
public final void writeFloat(float v) throws IOException {
    writeInt(Float.floatToIntBits(v));
}
```

Figure 6.7: Identical code in output stream classes.

```
// java.io.DataOutputStream
public final void writeInt(int v) throws IOException {
    OutputStream out = this.out;
    out.write((v >>> 24) & 0xFF);
    out.write((v >>> 16) & 0xFF);
    out.write((v >>> 8) & 0xFF);
    out.write((v >>> 0) & 0xFF);
    incCount(4);
}
```

Figure 6.8: Method `writeInt()` in `DataOutputStream`.

6.4.2 Output Stream Classes

Classes `RandomAccessFile` and `DataOutputStream` have some identical methods (for example, method `writeFloat()` in Figure 6.7).

In addition, there are several situations where the code in `DataOutputStream` can be made identical to the code in `RandomAccessFile` (using an abstraction), except for some extra lines of code following the identical part. We can promote all such code to the common superinterface and make a super call to it from the type which contains the extra lines of code.

For example, Figure 6.8 and Figure 6.9 show the method `writeInt()` from classes `DataOutputStream` and `RandomAccessFile` respectively. Figure 6.10 shows the common abstracted method that has been promoted to interface `DataOutput`. There is no method for `writeInt()` in `RandomAccessFile`. However, Figure 6.11 shows the method that remains in `DataOutputStream` to make the super call and perform the extra action. The `super(DataOutput)` is not standard Java. It is the super call to a superinterface, discussed throughout this dissertation. In fact, only the methods from `DataOutputStream` have to provide both an abstraction and a super call. The methods from `RandomAccessFile` only need to be abstracted. Therefore, they are completely

```
// java.io.RandomAccessFile
public final void writeInt(int v) throws IOException {
    this.write((v >>> 24) & 0xFF);
    this.write((v >>> 16) & 0xFF);
    this.write((v >>> 8) & 0xFF);
    this.write((v >>> 0) & 0xFF);
}
```

Figure 6.9: Method `writeInt()` in `RandomAccessFile`.

```
// java.io.DataOutput
public final void writeInt(int v) throws IOException {
    Sink out = this.sink();
    out.write((v >>> 24) & 0xFF);
    out.write((v >>> 16) & 0xFF);
    out.write((v >>> 8) & 0xFF);
    out.write((v >>> 0) & 0xFF);
}
```

Figure 6.10: Code abstracted in `DataOutput` interface.

promoted to `DataOutput`.

We ran test programs that used the re-designed `java.io` library partially shown in Figure 6.1. In Table 6.2 and Table 6.3 we show how multiple code inheritance reduces the amount of identical and similar code to simplify program construction and maintenance. Table 6.2 shows the number of methods that could be promoted in these stream classes of the `java.io` library, if Java supported multiple code inheritance. Table 6.3 shows the number of lines of executable code moved to the superinterfaces using the same multiple code inheritance assumption. We counted only executable lines and declarations, not comments or method signatures.

```
// java.io.DataOutputStream
public final void writeInt(int v) throws IOException {
    super(DataOutput).writeInt(v); // Proposed syntax.
    incCount(4);
}
```

Figure 6.11: Re-factored code in `DataOutputStream` with both abstraction and super.

Class	Identical methods	Abstract	Abstract and Super	Total promoted	Method Decrease
DataInputStream	4/19	8/19	0/19	12/19	63%
DataOutputStream	2/17	0/17	6*/17	2+6*/17	12%
RandomAccessFile	6/45	8/45	6/45	20/45	44%

Table 6.2: Method promotion in the Java stream classes using multiple code inheritance.

Class	Initial Lines	Added Lines Abstract and Super	Net Lines Abstract and Super	Line Decrease
DataInputStream	127	1	84	34%
DataOutputStream	83	7	66	20%
RandomAccessFile	154	2	97	37%

Table 6.3: Lines of code promotion in the Java stream using multiple code inheritance.

More important than the size of the reductions is the reduced cost of understanding and maintaining the abstracted code. Even though most of the method bodies of six methods move up from `DataOutputStream` to `DataOutput`, small methods remain that make super calls to these promoted methods. This is the reason that the method decrease is smaller for `DataOutputStream` than its code decrease. Reducing the number of lines of code reduces the maintainance cost for this code and enhances readability for users of this code.

These re-factored library classes exercise all of the multiple code inheritance implementation changes that we made. The test programs ran without error and with negligible time penalties for multiple-code inheritance.

Our test program (which uses the re-factored types) creates an instance of `DataOutputStream` which is sent `write` messages (`writeDouble()`, `writeInt()`, `writeChar()`, and `writeChars()`) in order to create an output text file and write some values in it. Then a `DataInputStream` object is created which uses the same file to read information for it (`readDouble()`, `readInt()`, `readChar()`, and `readLine()`). Although `DataInputStream` does not override any of the methods sent to a `DataInputStream` object (because these methods have been

promoted to `DataInput`), the program generates the same results as the unmodified `java.io` library.

6.5 Concluding Remarks

In this Chapter, we described several experiments we conducted to validate our JVM changes, targeting both single and multiple inheritance programs. The results of the tests and experiments show that our multiple code inheritance implementation preserves semantics and performance of existing single inheritance code, without altering Java language syntax or Java compilers. The dispatch scenarios illustrated in Chapter 4 were implemented and ran correctly.

In addition, we showed that both our basic multiple code inheritance and the super call mechanism that we implemented execute correctly in multiple inheritance programs. We also described how all the dispatch scenarios illustrated in Chapter 5 were implemented and ran without error, generating correct results.

Finally, we provided some measurements of the software engineering advantages of using multiple code inheritance. In order to test multiple inheritance programs, we used the re-factored `java.io` library, with code in interfaces and super calls to interface code. By using multiple code inheritance, a considerable amount of executable code was promoted to common super-types by being removed from the base type or replaced with only a super call.

Chapter 7

Syntax Support for Compilation

The ability to support multiple-inheritance of code introduces two specific challenges to the compilation process. First, as discussed earlier, current Java compilers do not support executable code inside interfaces. Second, a mechanism is needed to handle generalized super calls. Future work will be to modify the compiler to support both code in interfaces and the super call mechanism.

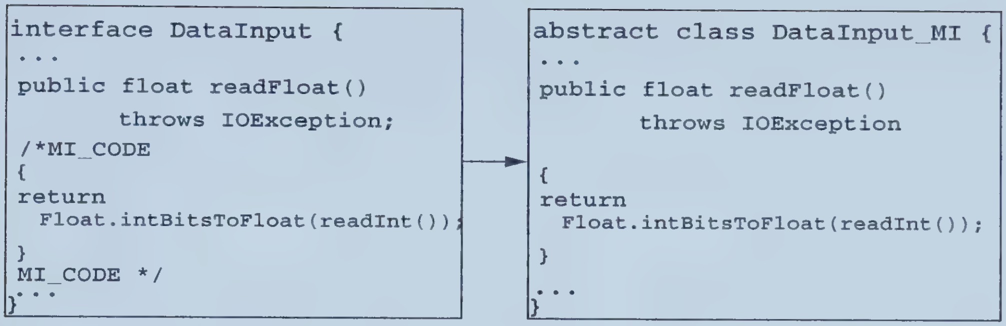
7.1 The Scripting Process

We have developed a translation process that uses an unmodified Java compiler and does not affect the existent language syntax. Our technique is based on source-to-source and class-file-to-class-file transformations using custom scripts, publicly available Java tools, and syntactic conventions in the user's Java code. All of our scripts have the prefix "MI_" (multiple-inheritance) in their names. Although there are several steps in the compilation process, the process is automated and it is summarized in a flow chart in Figure 7.3.

At the programmer level, the process is the following:

1. The programmer includes code in the interface, but the code is within comments with a special label `MI_CODE`.

2. Our scripts transform the `.java` file for the interface into a `.class` file that contains the code. We make use of the following tools: `jasper` [16] and `jasmin` [15].



```
interface DataInput {
 ...
 public float readFloat()
        throws IOException;
 /*MI_CODE
 {
 return
   Float.intBitsToFloat(readInt());
 }
 MI_CODE */
 ...
}
```

```
abstract class DataInput_MI {
 ...
 public float readFloat()
        throws IOException

 {
 return
   Float.intBitsToFloat(readInt());
 }
...
}
```

Figure 7.1: Syntax of interface code in `java.io.DataInput` interface and the result of applying the script `MI_hybridInterface`.

3. Multi-inheritance super calls are written as two standard Java instructions and our scripts translate them into the `invokeinterface` bytecodes described in Chapter 5.

7.2 Code in Interfaces

Current Java compilers do not allow code to be included in interfaces so the programmer delimits the code using special comment delimiters `/* MI_CODE` and `MI_CODE */`. For example, consider the interface `DataInput` and the class `DataInputStream` from Figure 3.14. The code for `readFloat()` in the interface `DataInput` is shown in Figure 7.1.

The goal of our compilation process is to create a file `DataInput.class` with Java bytecodes for the body of the method `readFloat()`, i.e., an interface with code. This is accomplished by creating both an interface `DataInput` and a class with the same name followed by `_MI` (i.e., multiple inheritance), then combining the `.class` files of both the interface and class into a single `.class` file that is like an interface, except that it contains code from the specially commented methods. Therefore step 2 of our process is divided into sub-steps that use several translation tools and scripts.

- 2.1 The interface source file (`DataInput.java` of Figure 7.1) is compiled using a standard `javac` compiler to create a binary file (`DataInput.class`) for the interface that contains no code.

- 2.2 The interface binary file (`DataInput.class`) is disassembled, using the `jasper` [7] tool into an interface jasper file (`DataInput.j`). The jasper file is a human-readable form of the binary file that begins with a description indicating that the file was originally compiled from an interface.

- 2.3 Script `MI_hybridInterface` performs a source-to-source translation from an interface source file (`DataInput.java`) into an abstract class source file (`DataInput_MI.java`) in which the special comment delimiters are removed from the interface's methods (`readFloat()`), making the code visible to a compiler. The class `DataInput_MI` is made abstract to avoid irrelevant compiler error messages since some interface methods may not contain code and a class that contains at least one method without code (abstract method) should be declared abstract.

- 2.4 The abstract class source file (`DataInput_MI.java`) is compiled by `javac` into an abstract class binary file (`DataInput_MI.class`) that contains code for all of the methods in the original interface that had methods (`readFloat()`).

- 2.5 The abstract class binary file (`DataInput_MI.class`) is disassembled into an abstract class jasper file (`DataInput_MI.j`) using the `jasper` tool.

- 2.6 Script `MI_copyHeaderInterface` first replaces all the `invokevirtual` bytecodes whose static types have a suffix `_MI` with `invokeinterface` bytecodes. Due to the difference in the number of operands required by `invokevirtual` (only two operands) and `invokeinterface` (four operands), another bytecode which represents the number of arguments taken by the method has to be added at the new `invokeinterface` location. This number is actually ignored at run-time, since the actual number of arguments is taken from the resolved `methodblock`. However, some number must be placed in the operands in order to allow the generation of the modified `.class` file with `jasmin`. The fourth operand is set

to zero by default, so we do not have to explicitly provide it. Recall that this step is necessary, because of the situation when we have a call-site `this.alpha()` within an interface method.

In this case, the script removes the `_MI` suffixes of all references in the abstract class jasper file (`DataInput_MI.j`). Note that although the static type of the receiver at an `invokevirtual` call-site is a class (`DataInput_MI`), after removing the `_MI` suffix and replacing the opcode with `invokeinterface`, the static type of the receiver at the same call-site becomes an interface (`DataInput`) as expected for an `invokeinterface` bytecode. The script combines this modified abstract class jasper file (`DataInput_MI.j`) with the interface jasper file (`DataInput.j`) to obtain a hybrid jasper file that has the header (description of the type of the `.class` file) of an interface (`DataInput.j`) and the code for the methods (`DataInput_MI.j`), except for the constructors. The hybrid jasper file overwrites the interface jasper file (`DataInput.j`).

- 2.7 The hybrid jasper file (`DataInput.j`) is assembled into a hybrid binary file (`DataInput.class`) using the `jasmin` [8] tool. Since `jasmin` is not a full-fledged compiler, it does not explicitly check whether or not interfaces have code so no errors are reported.

Although there are seven steps in this process, they are hidden from the programmer who uses the simple syntax of Figure 7.1. For now, all the steps of the process are automated in a `makefile`, therefore the user only types the `make` command. In the future, we would like the user to run a script instead of a `makefile` in order to trigger the execution of this process. Currently, when a program that has code in interfaces is executed by `java`, the verifier must be turned off (`-noverify` flag). We plan to modify our JVM to remove only the verification code for interfaces so the rest of verification can be maintained.

```
// Multi-super:  the lookup starts from the argument's IMT,
// continuing along its superinterface hierarchy.
    MI.supercall("InterfaceH");
    super.alpha();
    ...
// Normal super:  the lookup starts in the superclass MT,
// continuing along its superclass chain.
    super.alpha();
```

Figure 7.2: Syntax of `supercall` for call-sites in `ClassM`.

7.3 Super Calls

In Chapter 5, we described multi-inheritance super calls and introduced the syntax `super(Start).alpha()`, indicating the interface `Start` as the place the lookup begins from. Our approach currently uses two standard Java statements to represent this language extension. This allows us to still use the standard Java compiler, `javac`, albeit as part of a multi-step, scripted compilation process. To make a multi-inheritance super call, the programmer inserts a special static method call that contains the start interface as an argument, followed by a standard local method call. For example, Figure 7.2 shows the current syntax for the super calls shown in Figure 5.2 that start searching in `InterfaceH` (multi-inheritance super) and `ClassL` (normal super), respectivelly. `MI` is a new library class specifically designed to provide syntax support for multiple-inheritance. It can be discarded once compiler support is developed for multiple-inheritance using `super(Start).alpha()`. The `MI` class contains a static method `supercall` that takes as an argument the interface from which the lookup starts. This is a marker which indicates that the super call immediately following it is a special super, i.e., a multi-inheritance super call.

Since we do not alter the semantics of the existing super calls, we do not provide an `MI.supercall` statement before a normal super to a class. Thus we do not impose any overhead on existing super calls.

If the `javac` compiler tries to compile the code in Figure 7.2, based on

the inheritance hierarchy of Figure 5.2, it will produce a compilation error for both `super.alpha()` call-sites. In each case, it will search the superclass chain of `ClassM`, starting with `ClassL` and will not find a declaration for `alpha()`. To avoid spurious compilation errors, we can replace the `super` keyword with `this` for all the call-sites immediately preceded by an `MI.supercall` before compilation. This works since if the call-site `super.alpha()` is turned into `this.alpha()`, the compiler finds the method in the virtual method table (VMT) of the current class, therefore it does not report an error. However, an `invokevirtual` bytecode is generated instead of an `invokespecial`. We further need to replace this `invokevirtual` with an `invokeinterface`, so that the lookup starts in the IMT of the specified interface, and not in the method table (MT) of the superclass of the class which contains the super call-site.

Here is our multi-step compilation process that translates the syntax of Figure 7.2, to the bytecodes described in Chapter 5. These steps are the sub-steps of step 3 of the high-level compilation process presented at the beginning of Chapter 5. In each step, the term current class refers to the class that contains the super call. The example used is the code in Figure 7.2 with the inheritance hierarchy of Figure 5.2.

- 3.1 Script `MI_preprocessClass` transforms the current class source file (`ClassM.java`) into an abstract class source file (`ClassM_MI.java`) by adding the abstract modifier to the class. At the same time, the `super` keyword is replaced by the `this` keyword at all the call-sites immediately preceded by `MI.supercall`. The abstract modifier is needed since the current class may not actually declare the method invoked by the super call. For example, consider the situation where the code in Figure 7.2, is in a method called `beta()` and there is no code for `alpha()` in `ClassM`. By making the current class (`ClassM`) abstract, no compiler error will be generated by the `this.alpha()` call, because a slot for `alpha()` is automatically created in the virtual method table (VMT) of `ClassM`, representing a Miranda Method (detailed in Chapter 4).

- 3.2 The abstract class source file (`ClassM_MI.java`) is compiled into an

abstract class binary file (`ClassM_MI.class`) using `javac`.

- 3.3 The abstract class binary file (`ClassM_MI.class`) is disassembled into an abstract class jasper file (`ClassM_MI.j`) using `jasper`.

- 3.4 The script `MI_abstractToConcrete` translates the abstract class jasper file (`ClassM_MI.j`) into a concrete class jasper file (`ClassM.j`). The abstract class modifier is removed and the `invokevirtual` instruction after the `MI.supercall(Start)` method invocation is changed to an `invokeinterface` instruction. The argument of the `MI.supercall` is copied over the static type of the receiver in the `invokeinterface` immediately following this statement. As in all cases where the `invokevirtual` bytecode was replaced with an `invokeinterface` requiring two more operands, the number of arguments is also supplied. Since the number of arguments is ignored at run-time, being retrieved from the resolved methodblock, we can use it as a marker for the multi-super case, setting it to 255. Now the modified `.j` file can be correctly generated with `jasmin` resulting in a valid `.class` file, since the `invokeinterface` has been provided with the number of operands it requires.

- 3.5 The concrete class jasper file (`ClassM.j`) is assembled into a concrete class binary file (`ClassM.class`) using `jasmin`.

The same process works on an interface source file that contains a super call. Although there are five steps in this process, they are hidden from the programmer who uses the simple syntax of Figure 7.2.

7.4 Concluding Remarks

In this Chapter, we presented our scripting process that was developed to cope with the absence of compiler support for multiple code inheritance.

We solved two specific challenges to the compilation process. First, as discussed earlier, current Java compilers do not support executable code inside interfaces. Second, a mechanism is needed to handle generalized super calls.

The proper way to solve these problems is to modify a compiler to support our changes and we plan to complete this task in the future. In the meantime, we prototyped the compiler, through the scripting process described in this Chapter. Our scripting process works with any existing java compiler.

Although there are several steps in this scripting process, they are automated and the user only executes a `makefile` to trigger their execution. In the future, we would like to have a script with the same functionality as the current `makefile`.

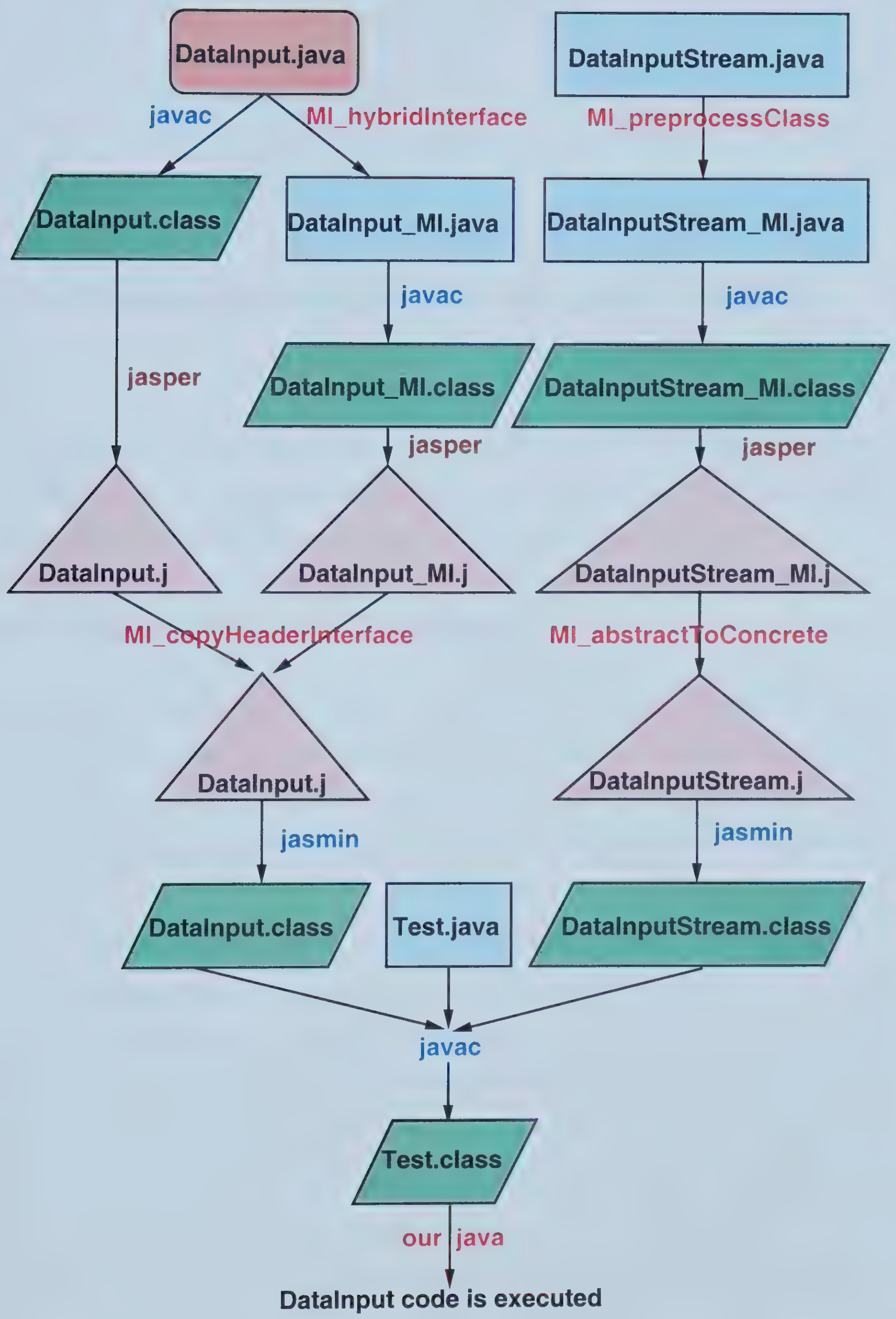


Figure 7.3: The scripting process.

Chapter 8

Conclusions and Future Work

In this dissertation we presented the design and implementation of an extended JVM that supports multiple code inheritance. We conclude with a summary of Chapters, future directions and research contributions.

8.1 Summary of Chapters

We started by motivating the need for multiple code inheritance in Java, emphasizing its advantages: facilitates code re-use, supports separation of inheritance concepts, and improves expressiveness and clarity of implementation. Moreover, multiple code inheritance avoids duplicated code and supports refactoring.

We continued with a short review of the current state of multiple inheritance, investigating the mechanisms of multiple code inheritance in several programming languages. We support multiple code inheritance, and not multiple data inheritance, since the latter is not as important as code inheritance. Multiple data inheritance is not a popular feature among programming languages which support multiple inheritance, being the source of many complications. Re-using code is a powerful object-oriented feature which decreases the effort of programmers, who are mainly focused on implementing method bodies.

We described the current implementation of those parts of the JVM involved in method dispatch. The steps of method dispatch, resolution and execution, are detailed for the `invokevirtual` and `invokeinterface` bytecodes.

Since resolution is slow, bytecode quicking is introduced.

We proposed a mechanism to support multiple code inheritance in Java through code in special interfaces that represent code-types. Then we described how we modified the JVM loader to support these special types and showed how the code was dispatched. We also described our solution to the dispatch of `this.alpha()` call-sites within interface methods.

We presented the changes necessary to support a generalization of the `super` operation for multiple inheritance. We defined and implemented a super call mechanism that resembles the one in C++. We implemented this by making a dispatch time change to the virtual machine. We provided a comment notation for including code in interfaces and a simple notation for super calls to interfaces that does not require compiler support. We proposed syntax changes for super calls to interfaces that would simplify coding and would require future compiler modification.

We conducted several experiments to validate our approach, targeting both single and multiple inheritance programs. The dispatch scenarios illustrated in Chapter 4 and Chapter 5 were implemented and ran correctly for both the basic multiple code inheritance and our generalized super call implementations.

The multiple inheritance test programs used the re-factored `java.io` library hierarchy, in which interface code and our generalized super calls to interfaces are correctly dispatched. The measurements of the software engineering advantages of using multiple code inheritance show that a considerable amount of executable code is promoted to common super-types, being either removed from the base types or replaced with a super call.

Finally, we discussed the scripting process we used in order to insert code into interfaces and to support super calls to interfaces, since the compiler is not modified. We proposed syntax changes to simplify this mechanism in the perspective of a modified compiler which accepts code within interfaces.

[illegible]

8.2 Future Work

In this Section, we mention several ideas which, if expanded, can contribute to the improvement of our JVM.

1. Even though the changes we have completed in order to support super calls are small and localized, it is more appropriate to provide a new bytecode for the multiple-inheritance super calls, namely `invokemulti-super`. We would like to add this bytecode to our JVM and further evaluate its performance. Alternately, we could mark the `invokeinterface` bytecode using code attributes. Other researchers have successfully used code attributes to mark bytecodes [27].

2. Currently, in order to compile code in interfaces, we execute a set of scripts. We plan to change this in the future by modifying a compiler to support the `super(InterfaceA)` syntax in Java, which would make our scripting process unnecessary.

3. We also plan to modify our JVM to support the verification of code in interfaces, at the same time maintaining the rest of the verification stages.

4. We would like to validate the portability of our modifications to a different JVM which supports a JIT compiler.

5. In addition, we look for other opportunities to re-factor type hierarchies by using our modified JVM, evaluate the decrease in code that we could achieve and measure the performance differences.

8.3 Research Contributions

The research contributions of this dissertation include:

1. The first implementation of multiple code inheritance in Java is provided. It is based on the novel concept of adding code to a new type of interface, called a code-type. No changes need to be made to the syntax of Java

to use multiple code inheritance, so no compiler changes are necessary. However, syntax changes that would simplify coding are proposed for the future.

2. We show how multiple code inheritance reduces the amount of identical and similar code (such as in the standard libraries) to simplify program construction and maintenance. We re-factor the `java.io` library and show that programs using the classes in this library run correctly.

3. We define and implement a super call mechanism that resembles the one in C++, in which programmers can specify an inheritance path to the desired superinterface (code-type) implementation. We introduce a simple notation for these super calls that does not require compiler support and propose a simple syntax for future compiler support.

Our modifications are small and localized. The changes consist of:

1. The changes to algorithm `ConstructIMT` executed by the class loader as shown in Chapter 4.

2. The changes to execution of the `invokeinterface_quick` bytecode to recognize a marked `invokemulti-super` that are shown in Chapter 5.

Our approach facilitates code re-use, reducing the amount of code that the programmer has to write, supports separation of inheritance concepts, and improves expressiveness and clarity of implementation. Existing Java compilers, libraries and programs are not affected by our JVM modifications and single-inheritance programs can achieve performance comparable to the original JVM. Moreover, execution of multiple inheritance programs is correct, for both our basic multiple code inheritance implementation and the super call mechanism.

Bibliography

[1] Gilad Bracha and William Cook. Mixin-based Inheritance. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'90)*, pages 303–311, Ottawa, Canada, October 1990. ACM Press.

[2] Timothy Budd. *An Introduction to Object-Oriented Programming, Second Edition.* Addison-Wesley, 1997.

[3] Cecil. http://www.cs.washington.edu/research/projects/cecil/.

[4] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, volume 35(10), pages 130–145, Minneapolis, USA, 2000.

[5] Clos. http://www-2.cs.cmu.edu/groups/ai/html/cltl/cltl2.html.

[6] Brad Cox and Andrew J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach, Second Edition.* Addison-Wesley, 1986.

[7] L. Peter Deutsch and Alan Schiffman. Efficient Implementation of the Smalltalk-80 System. *In Principles of Programming Languages*, 1994.

[8] C. Dutchyn, P. Lu, D. Szafron, S. Bromling, and W. Holst. Multi-Dispatch in the *Java Virtual Machine*: Design and Implementation. In *Proceedings of 6th Usenix Conference on Object-Oriented Technologies and Systems (COOTS'2001)*, pages 77–92, San Antonio, USA, January 2001.

[9] Chris Dutchyn. Multi-Dispatch in the *Java Virtual Machine*: Design, Implementation, and Evaluation. Master's thesis, Department of Computing Science, University of Alberta, 2002.

[10] Dylan. http://www.dylanworld.com/dylan_reference.html.

[11] Eiffel. http://docs.eiffel.com/.

[12] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, New Jersey, 1990.

[13] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, 2nd Edition.* Addison-Wesley Publishing Company, Reading, Massachusetts, 2000.

[14] W. Holst and D. Szafron. A General Framework for Inheritance Management and Method Dispatch in Object-Oriented Languages. In *Proceedings of the Object-Oriented Programming 11th European Conference (ECOOP'97), Lecture Notes in Computing Science 1241, Springer-Verlag*, pages 276–301, Finland, June 1997.

[15] Jasmin. http://www.mrl.nyu.edu/ meyer/jvm.

[16] Jasper. http://www.angelfire.com/tx4/cus/jasper.

[17] W. R. LaLonde and J. Pugh. *Subclassing $\neq$ subtyping $\neq$ is $-$ a. Journal of Object-Oriented Programming*, 3(5):57–62, 1991.

[18] E language. http://www.erights.org/elang/intro/.

[19] Y. Leontiev, M. T. Özsu, and D. Szafron. On Separation between Interface, Implementation and Representation in Object DBMSs. In *Proceedings of the 26th Technology of Object-Oriented Languages and Systems Conference (TOOLS USA98)*, pages 155–167, Santa Barbara, USA, August 1998.

[20] Yuri Leontiev. *Type System for an Object-Oriented Database Programming Language.* PhD thesis, Department of Computing Science, University of Alberta, 1999.

[21] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the *Java Virtual Machine.* In *Proceedings of the 13th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98), SIGPLAN Notices*, volume 33, pages 36–44, Vancouver, Canada, October 1998. ACM Press.

[22] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, 2nd Edition.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1999.

[23] M. Mohnen. Interfaces with Skeletal Implementations in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00), Poster Session*, Cannes, France, June 12th - 16th 2000. http://www-i2.informatik.rwth-aachen.de/ mohnen/PUBLICATIONS/ecoop00poster.html.

[24] C. Pang, W. Holst, Yuri Leontiev, and D. Szafron. Multi-Method Dispatch Using Multiple Row Displacement. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99)*, pages 304–328, Lisbon, Portugal, June 1999.

[25] Perl. http://www.perl.com/.

[26] Sun Microsystems Inc. Java[tm] 2 Platform. http://www.sun.com/software/communitysource/java2/download.html.

[27] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A Framework for Optimizing Java Using Attributes. *Lecture Notes in Computer Science*, 2027:334, 2001.

[28] Python. http://www.python.org/.

[29] Sather. http://www.icsi.berkeley.edu/ sather/.

[30] Yen-Ping Shan, Tom Cargill, Brad Cox, William Cook, Mary Loomis, and Alan Snyder. Is Multiple Inheritance Essential to OOP? In *OOPSLA '93, ACM Sigplan Notices*, volume 28(10), pages 360–363, October 1993.

[31] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[32] C. Szypersky, S. Omohundro, and S. Murer. Engineering a Programming Language: The Type and Class System of Sather. Technical report TR-93-064, The International Computer Science Institute, November 1993.

[33] A. Taivalsaari. On the Notion of Inheritance. *ACM Computing Surveys*, 28(3):439–479, September 1996.

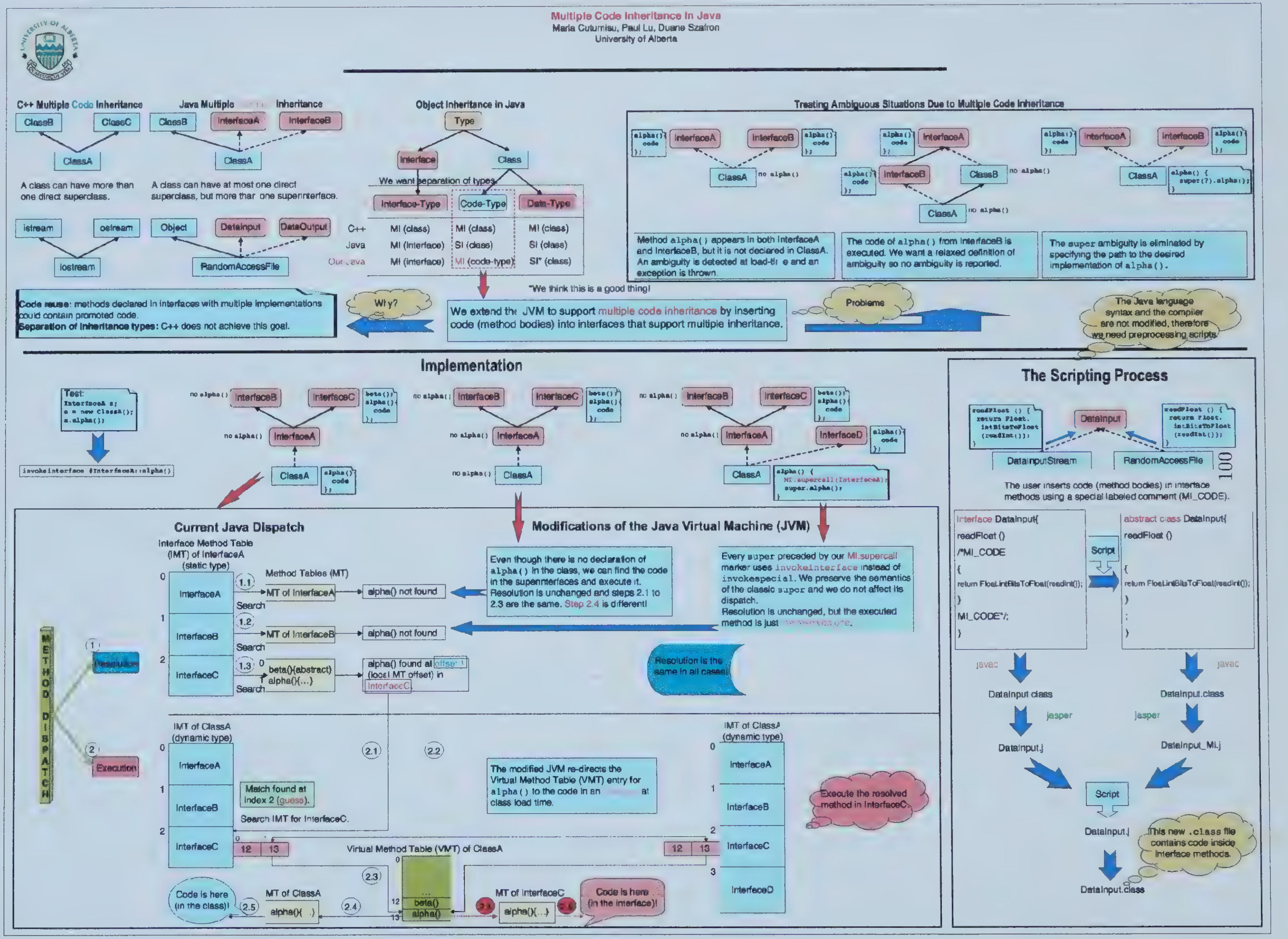
[34] Bill Venners. *Inside the Java Virtual Machine , Second Edition.* McGraw-Hill Osborne Media, 2000.

[35] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'87)*, pages 168–182, Orlando, USA, October 1987.

Appendix A

Dissertation Highlights

This Appendix Section illustrates the most important parts of our implementation of multiple code inheritance in Java.



Multiple Code Inheritance in Java

Maria Cutumisu, Paul Lu, Duane Szafron
University of Alberta

A class can have more than one direct superclass.

A class can have at most one direct superclass, but more than one superinterface.

*We think this is a good thing!

Code reuse: methods declared in interfaces with multiple implementations could contain promoted code.
Separation of inheritance types: C++ does not achieve this goal.

We extend the JVM to support multiple code inheritance by inserting code (method bodies) into interfaces that support multiple inheritance.

Method `alpha()` appears in both InterfaceA and InterfaceB, but it is not declared in ClassA. An ambiguity is detected at load-ti e and an exception is thrown.

The code of `alpha()` from InterfaceB is executed. We want a relaxed definition of ambiguity so no ambiguity is reported.

The `super` ambiguity is eliminated by specifying the path to the desired implementation of `alpha()`.

The Java language syntax and the compiler are not modified, therefore we need preprocessing scripts.

Implementation

Test:
InterfaceA a;
a = new ClassA();
a.alpha();
invokeinterface #InterfaceA::alpha()
InterfaceA
InterfaceB
InterfaceC
InterfaceD
ClassA
no alpha()
beta(); alpha() { code };
alpha() { MI.supercall(InterfaceA); super.alpha(); }

Current Java Dispatch

METHOD DISPATCH
1 Resolution
2 Execution
Interface Method Table (IMT) of InterfaceA (static type)
0 InterfaceA
1 InterfaceB
2 InterfaceC
Method Tables (MT)
1.1 MT of InterfaceA
alpha() not found
Search
1.2 MT of InterfaceB
alpha() not found
Search
1.3 0 beta(){abstract} alpha(){...}
Search
alpha() found at offset 1 (local MT offset) in InterfaceC
IMT of ClassA (dynamic type)
0 InterfaceA
1 InterfaceB
2 InterfaceC
12 13
Match found at index 2 (guess).
Search IMT for InterfaceC.
2.1
2.2
2.3
2.4
2.5
Virtual Method Table (VMT) of ClassA
0 ...
12 beta()
13 alpha()
MT of ClassA
alpha(){...}
Code is here (in the class)!
MT of InterfaceC
alpha(){...}
Code is here (in the interface)!
IMT of ClassA (dynamic type)
0 InterfaceA
1 InterfaceB
2 InterfaceC
3 InterfaceD
12 13
Execute the resolved method in InterfaceC.
Resolution is the same in all cases!

Modifications of the Java Virtual Machine (JVM)

Even though there is no declaration of `alpha()` in the class, we can find the code in the superinterfaces and execute it. Resolution is unchanged and steps 2.1 to 2.3 are the same. Step 2.4 is different!

Every `super` preceded by our MI.supercall marker uses `invokeinterface` instead of `invokespecial`. We preserve the semantics of the classic `super` and we do not affect its dispatch.
Resolution is unchanged, but the executed method is just [illegible].

The modified JVM re-directs the Virtual Method Table (VMT) entry for `alpha()` to the code in an [illegible] at class load time.

The Scripting Process

The user inserts code (method bodies) in interface methods using a special labeled comment (MI_CODE).

```
Interface DataInput{
readFloat ()
/*MI_CODE
{
return Float.intBitsToFloat(readInt());
}
MI_CODE*/;
}
```

```
abstract class DataInput{
readFloat ()
{
return Float.intBitsToFloat(readInt());
}
;
}
```